SUELENI MENDEZ BATISTA

UMA FERRAMENTA DE APOIO À DEFINIÇÃO DE REQUISITOS DA *MDSODI* NO CONTEXTO DO AMBIENTE *DiSEN*

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática pelo Curso de Pós-Graduação em Informática, do Setor de Ciências Exatas da Universidade Federal do Paraná, em convênio com o Departamento de Informática da Universidade Estadual de Maringá.

Orientadora: Dr.ª Elisa H. Moriya Huzita

CURITIBA
2003

Ministério da Educação
Universidade Federal do Paraná
Mestrado em Informática

PARECER

Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, da aluna *Sueleni Mendez Batista* , avaliamos o trabalho intitulado, *"Uma Ferramenta de Apoio à Definição de Requisitos da MDSODI no Contexto do Ambiente DiSEN"*, cuja defesa foi realizada no dia 27 de agosto de 2003, às quatorze horas, no Auditório do Departamento de Informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação da candidata. (Convênio número 279-00/UFPR de Pós-Graduação entre a UFPR e a UEM - ref. UEM número 1331/2000-UEM).

Curitiba, 27 de agosto de 2003.

Profª. Dra. Elisa Hatsue Moriya Huzita
DIN/UEM – Orientadora

Profª. Dra. Rosângela Aparecida Dellosso Penteado
DC/UFSCAR – Membro Externo

Prof. Dr. Alexandre Ibrahim Direne
DINF/UFPR – Membro Interno

AGRADECIMENTOS

A Deus, que sempre me acompanhou, me deu saúde e força para realizar este trabalho.

Aos meus pais Paulo e Clarice, pelo amor e educação recebidos durante toda a minha vida.

A meu irmão Denerval, pelo apoio e companheirismo demonstrados durante a realização desse curso de pós-graduação. Dê, sem seu apoio seria impossível (cp).

A minha orientadora Elisa Hatsue Moriya Huzita, pelo apoio e paciência durante a orientação, pelo respeito, profissionalismo e companheirismo demonstrados durante o período em que trabalhamos juntos.

Aos amigos de curso Cesar Fernando Moro, Maria Edith Vilella Pedras e Luiz Vinicius Ribas, pelas contribuições, paciência e força. À amiga de projeto Márcia Cristina Dadalto Pascutti pela força. Ao Edson Alves de Oliveira Junior e Marco Aurélio pelo apoio em Java.

Às amigas do DIN-UEM Maria Madalena Dias e Tânia Fátima Calvi Tait que sempre estiveram ao meu lado durante essa caminhada.

Aos amigos do DEQ-UEM, pelo apoio recebido e pela compreensão nos momentos de ausência, a amiga Célia, pelo empréstimo do *notebook*, a amiga Elenice pelo companheirismo, ao amigo Lúcio pelo conselhos e ao amigo Dorival pelo suporte ao meu trabalho.

Aos demais amigos que, de maneira direta ou indireta, contribuíram para a realização deste trabalho e torceram pelo meu sucesso.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

ADS	– Ambiente de Desenvolvimento de *Software*
BMV	– *Basic Model View*
CASE	– *Computer Aided Software Engineering*
CREWS	- *Cooperative Requirements Engineering With Scenarios*
DART	– *Distributed Artefact Repository*
DIMANAGER	–*Distributed Software Manager*
DIN	– Departamento de Informática
DiSEN	– *Distributed Software Engineering Environment*
I*	– *Distributed Intentionality*
JAD	– *Joint Application Design*
JMI	– *Java Metadata Interface*
KAOS	– *Knowledge Acquisition in AutOmated Specification*
LEL	– *Language Extended Lexicon*
LES	– Laboratório de Engenharia de *Software*
MDR	– *Metadata Repository*
MDSODI	– Metodologia para Desenvolvimento de *Software* Distribuído
MOF	– *Meta Object Facility*
MOOPP	- Metodologia OO para Desenvolvimento de *Software* Paralelo
OMG	– *Object Management Group*
SCRAM	- Método para a Análise de Requisitos Baseados em Cenários
UdI	– Universo de Informações
UEM	– Universidade Estadual de Maringá
UML	– *Unified Modeling Language*
XML	– *eXtensible Markup Language*

RESUMO

A crescente complexidade das aplicações, a contínua evolução tecnológica e o uso cada vez mais disseminado de redes de computadores têm estimulado os estudos referentes ao desenvolvimento de sistemas distribuídos. Sistemas distribuídos são bastante complexos, o que, conseqüentemente, reflete na complexidade de desenvolvimento do *software*. Para que o desenvolvimento de *software* distribuído seja uma tarefa produtiva, gerando também produtos de qualidade, é necessário que o ambiente de apoio ao desenvolvedor seja estruturado de modo a prover recursos que o auxiliem na realização do processo. Visando suprir a necessidade de ferramentas e ambientes de desenvolvimento de *software* distribuído, foram desenvolvidos a Metodologia para Desenvolvimento de *Software* Distribuído (*MDSODI)* e o ambiente *Distributed Software Engineering Environment (DiSEN). DiSEN* é um ambiente distribuído de desenvolvimento de *software*, no qual a *MDSODI* está inserida, que tem, como um de seus objetivos, permitir que vários desenvolvedores, atuando em locais distintos, possam trabalhar de forma cooperativa no desenvolvimento de *software*. O principal objetivo deste trabalho é o desenvolvimento de uma ferramenta de apoio à fase de requisitos da *MDSODI* no contexto do ambiente *DiSEN*. A ferramenta denominada *REQUISITE* apresenta um modelo de solução distribuída, baseada em cenários, independente de plataforma, onde vários *stakeholders* podem trabalhar de forma cooperativa, na fase de requisitos, no desenvolvimento de *software* distribuído.

ABSTRACT

The growing complexity of applications, and the constant technological progress and the massive use of computer network have stimulated the studies concerning the development of distributed systems. The distributed systems are very complex which, consenquently, reflects upon the complexity of the software development. In order to make the development of distributed software a productive task, resulting in quality products, it is necessary that the supporting environment be structured to provide resources that will help the developer to complete the process. Aiming at providing the necessary tools and supportive environment for distributed software the Methodology for the Development of Distributed Software (*MDSODI*) and the Distributed Software Engineering Environment (*DiSEN*) were developed. *DiSEN* is an environment of software development into which *MDSODI* is inserted and one of its objectives is to allow its several developers, working at different places, to perform their task cooperatively in the development of the software. The main objective of this dissertation is to develop a tool to support the requisites phase of the *MDSODI* in the context of the *DiSEN* environment. The tool called *REQUISITE* presents a model of distributed solution, based on scenarios and independent from platform, where several stakeholders can work cooperatively, in the requisite phase, on the development of distributed software.

1 INTRODUÇÃO

A crescente complexidade das aplicações, a contínua evolução tecnológica e o uso cada vez mais disseminado de redes de computadores têm impulsionado os estudos referentes ao desenvolvimento de sistemas distribuídos.

Segundo COULOURIS *et al.* (2001), sistemas distribuídos consistem em uma coleção de computadores autônomos ligados por uma rede, buscando, dessa forma, coordenar as atividades de maneira eficiente, além de propiciar o compartilhamento de recursos, quer sejam de *hardware* ou de *software*.

Os sistemas distribuídos se distinguem dos sistemas tradicionais, basicamente, devido às seguintes características: suporte aos recursos compartilhados, concorrência, tolerância a falhas e transparência (COULOURIS *et al.*, 2001). Sistemas distribuídos apresentam alguns problemas que, na maioria das vezes, chegam a ser mais complexos do que os que ocorrem com os sistemas tradicionais, e por isso precisam ser tratados de forma diferente. Dentre eles, podemos citar: problemas de comunicação, sincronização e quedas (COULOURIS *et al.*, 2001).

As características e os problemas apresentados mostram que sistemas distribuídos são bastante complexos, o que, conseqüentemente, reflete na complexidade de desenvolvimento do *software*. Neste contexto, para que o desenvolvimento de *software* distribuído seja uma tarefa produtiva, gerando também produtos de qualidade, é necessário que o ambiente de apoio ao desenvolvedor seja estruturado de modo a prover recursos (ferramentas, técnicas e metodologias) que o auxiliem na realização do processo.

Visando suprir a necessidade de ferramentas e ambientes de desenvolvimento de *software* distribuído, foram desenvolvidos a Metodologia para Desenvolvimento de *Software* Distribuído (*MDSODI*) (GRAVENA, 2000) e o ambiente *Distributed Software Engineering Environment* (*DiSEN*) (PASCUTTI, 2002). *DiSEN* é um ambiente distribuído de desenvolvimento de *software*, no qual a *MDSODI* está inserida, que tem, como um de seus objetivos, permitir que vários desenvolvedores,

atuando em locais distintos, possam trabalhar de forma cooperativa no desenvolvimento de *software*.

BROOKS (1987) declarou que a parte mais difícil da construção de um sistema de *software* é decidir precisamente o que construir. Nenhuma outra parte do trabalho conceitual é tão difícil como estabelecer os requisitos técnicos detalhados, incluindo todas as interfaces com pessoas, máquinas e outros sistemas de *software*. Nenhuma outra parte do trabalho afeta mais o sistema resultante se feita errada, e nenhuma outra parte é mais difícil de ser retificada depois.

A comunidade de engenharia de *software* concorda que os erros ocasionados durante a fase de requisitos estão entre os mais caros para se corrigir depois que o sistema está em funcionamento (KOTONYA; SOMMERVILLE, 1997), (WEIRINGA, 1996), (LOUCOULOS *et al.*, 1995), (PRESSMAN, 2001).

Do ponto de vista econômico, o custo da correção de erros de requisitos aumenta drasticamente à medida que eles são descobertos no decorrer das fases do desenvolvimento de *software*. Segundo PRESSMAN (2001), o custo de correção de erros na fase de projeto é cerca de três a seis vezes mais alto do que na fase de definição de requisitos. E o custo de correção de erros na fase de operação é cerca de 40 a 1000 vezes mais caro do que o custo da correção de erros na fase de definição dos requisitos.

Sendo assim, a fase de requisitos tem sido reconhecida como uma fase crítica do processo da engenharia de *software*. Esse reconhecimento decorre da descoberta de que a maior parte dos problemas, geralmente os mais dispendiosos e de maior impacto negativo no desenvolvimento de *software*, são originados nas etapas iniciais do desenvolvimento. Essas etapas constituem o processo de engenharia de requisitos, no qual as principais atividades podem ser definidas como: elicitação, análise e negociação, validação, documentação e gerenciamento de requisitos. Normalmente, falhas na realização dessas atividades resultam em documentos de requisitos inconsistentes, incompletos e, conseqüentemente, em produtos de *software* de baixa qualidade.

É importante salientar que o grande número de requisitos existentes em um sistema aponta para a necessidade de utilização de ferramentas *CASE* (*Computer Aided Software Engineering)*, as quais permitem controlar e facilitar o trabalho de engenheiros de requisitos (KOTONYA; SOMMERVILLE, 1997).

Nesse contexto, a implementação de uma ferramenta *CASE* de apoio à fase de requisitos no desenvolvimento de *software* distribuído é de grande valia. Segundo SOMMERVILLE (2001), as ferramentas *CASE,* se usadas adequadamente, proporcionam os seguintes benefícios: aumentam a produtividade, facilitam a prototipação, auxiliam na desvinculação entre modelagem e as decisões de implementação, facilitam também o desenvolvimento incremental de sistemas através da documentação e do re-projeto de sistemas, trazem redução do custo e do tempo de desenvolvimento de *software,* bem como o aumento da qualidade.

1.1 OBJETIVO

O principal objetivo deste trabalho é o desenvolvimento de uma ferramenta que dê suporte à Metodologia para Desenvolvimento de *Software* Distribuído (*MDSODI*), no ambiente distribuído de desenvolvimento de *software Distributed Software Engineering Environment* (*DiSEN*), na fase de requisitos, considerando aspectos de concorrência/paralelismo, distribuição, sincronização e comunicação. A ferramenta denominada *REQUISITE* deve auxiliar a concluir, com êxito, um acordo entre quem solicita e quem desenvolve *software*, estabelecendo, clara e rigorosamente, o que deverá ser produzido.

1.2 TRABALHOS RELACIONADOS

Dentre as ferramentas existentes atualmente e que apóiam a definição de requisitos, destacamos em ordem alfabética: *CaliberRM* (BORLAND, 2003), *DOORS* (TELELOGIC, 2003), *icCONCEPT RTM* (INTEGRATE, 2003), *RDT* (IGATECH, 2003), *RequisitePro* (RATIONAL, 2003) e *Slate Require* (EDS, 2003).

A ferramenta *CaliberRM* (BORLAND, 2003), da empresa *Borland Software Corporation*, é uma ferramenta colaborativa de gerenciamento de requisitos, de acesso via *Web*. A ferramenta *CaliberRM* utiliza cliente *Web* baseado em *Java*.

Disponível somente para *Windows*, a *CaliberRM* permite a comunicação através de grupos online de discussão e notificação automática das alterações via e-mail. Define terminologias através de um glossário online. Possui repositório centralizado de requisitos, rastreabilidade, análise de impacto e integração com todo o ciclo de vida de desenvolvimento. A ferramenta permite integração com as ferramentas *Borland StarTeam*, *Borland TestDirector* e *Microsoft Project*.

A ferramenta *DOORS/ERS* (*Dynamic Object-Oriented Requirement System / Enterprise Requirements Suíte*) (TELELOGIC, 2003), da empresa *Telelogic AB*, é um *suite* de gerenciamento de requisitos que captura, rastreia e gerencia informações. O *DOORS/ERS* é constituído da ferramenta de gerenciamento de requisitos *DOORS*, acesso baseado na Internet através da ferrementa *DOORSnet* e *DOORSrequireIT* para usuários que preferem trabalhar com o editor *Microsoft Word*. Utiliza um fluxo de comunicação estruturado para reduzir riscos de falhas nos processos de comunicação. Habilita o desenvolvimento colaborativo, tendo em comum a base de requisitos, um *Distributed Data Management* (*DDM*). Possibilita a validação visual, testando os objetos baseados nos requisitos propostos com as exigências definidas. Possui vários níveis de rastreabilidade como, por exemplo, requisitos de teste e requisitos de projeto.

As ferramentas *DOORS* e *DOORSnet* estão disponíveis para as plataformas *Windows* e *Unix*, enquanto a ferramenta *DOORSrequireIT* está disponível somente para plataforma *Windows*.

A ferramenta *icConcept RTM* (*Requirements & Traceability Management*) (INTEGRATE, 2003), da empresa *Integrate Chipware*, é um gerenciador de requisitos pertencente ao produto *RTM Workshop*, que tem por objetivo facilitar o processo e o gerenciamento de desenvolvimento em todo o ciclo de vida do projeto. É uma ferramenta de desenvolvimento distribuído, multi-usuários, arquitetura cliente-servidor, baseado no gerenciador de banco de dados *Oracle* e plataforma *Windows*.

A ferramenta *icConcept RTM* possibilita a análise de impacto, captura de requisitos, controle de versão e rastreabilidade através do ciclo de vida do projeto. Com o auxílio da ferramenta *icExplorer,* é possível visualizar e navegar por todos os dados do projeto. A captura de requisitos pode ser realizada com o auxílio das ferramentas *icWORD* e *icFRAME,* que são integrações com o *Microsoft Word* e com o *Adobe FrameMaker,* respectivamente. O acesso via *Internet* pode ser realizado através da ferramenta *icBrowser.*

A ferramenta *RDT* (*Requirements Design & Traceability*) (IGATECH, 2003), da empresa *Igatech Systems Pty Ltd,* foi desenvolvida para capturar e gerenciar requisitos. Permite acesso a múltiplos usuários via rede, rastreabilidade de requisitos, possui integração com o *Microsoft Word* e foi desenvolvido para plataforma *Windows.*

A ferramenta *RequisitePro* (RATIONAL, 2003), da empresa *Rational Corporation*, é utilizada para documentar e gerenciar requisitos durante todo o ciclo de vida de um projeto. Esta ferramenta é integrada com as demais ferramentas da *Rational* (*Rational Rose*, *Rational ClearQuest* e *Rational TestManager*), permitindo assim a integração dos requisitos com os demais artefatos do projeto.

A *RequisitePro* permite o trabalho de múltiplos usuários. O acesso via Internet é realizado com o auxílio da ferramenta *RequisiteWeb*, que é a interface *Web* da *RequisitePro*. Suporta os bancos de dados *Microsoft SQL* e *Oracle,* possui integração com o editor de textos *Microsoft Word* e está disponível somente para o ambiente *Windows*.

A ferramenta *Slate Require* (EDS, 2003), da empresa *Eletronic Data System Corporation* (EDS), é uma ferramenta de *groupware* para auxiliar no gerenciamento de requisitos. Essa ferramenta pertencente à família de produtos *SLATE* (*System Level Automation Tools for Enterprises*), que trabalham de forma integrada em um ambiente de *groupware* em todo o ciclo de vida de desenvolvimento.

Slate Require inclui um gerenciador de documentos que captura e gerencia requisitos de várias origens (documentos, e-mail, Internet, etc.) com completa rastreabilidade. Possui acesso via *Web* através de um portal denominado *tranSLATE* e

Teamcenter Requirements, o que possibilita a visualização, a criação e a edição de requisitos através do *Microsoft Word*, do *Excel* e do *Microsoft Project*.

A ferramenta apresentada neste trabalho difere das apresentadas acima por trazer uma solução independente de plataforma e por oferecer apoio a uma metodologia para desenvolvimento de *software* distribuído.

1.3 ESTRUTURA DO TRABALHO

Além do capítulo introdutório, este trabalho consiste de mais cinco capítulos.

No capítulo 2, são apresentados conceitos sobre requisitos e engenharia de requisitos. Em seguida, são descritos como os requisitos podem ser classificados, o processo da engenharia de requisitos e algumas técnicas de modelagem baseadas em cenários.

O capítulo 3 apresenta uma descrição do ambiente distribuído de desenvolvimento de *software DiSEN* e da metodologia para desenvolvimento de *software* distribuído *MDSODI*.

No capítulo 4, é apresentada a ferramenta *REQUISITE*, que dá suporte à *MDSODI* no ambiente distribuído de desenvolvimento de *software DiSEN*.

No capítulo 5, é apresentado um estudo de caso para a validação da ferramenta *REQUISITE*.

Finalmente, o capítulo 6 apresenta as conclusões, as contribuições deste trabalho, bem como os trabalhos em andamento e as propostas para trabalhos que possam vir a ser desenvolvidos em continuidade a este.

2 ENGENHARIA DE REQUISITOS

A engenharia de requisitos é uma sub-área da engenharia de *software*. A área surgiu em 1993, com a realização do I *International Symposium on Requirements Engineering*. Entender as necessidades e atender as expectativas dos clientes sempre foram colocados como alguns dos desafios da engenharia de *software*. A engenharia de requisitos tem por objetivo prover ao engenheiro de *software* métodos, técnicas e ferramentas que auxiliem o processo de compreensão e de registro dos requisitos que o *software* deve atender. Diferentemente de outras sub-áreas da engenharia de *software*, a área de requisitos tem que lidar com conhecimento interdisciplinar, envolvendo, muitas vezes, aspectos de ciências sociais e ciência cognitiva.

A etapa de requisitos é essencial ao processo de desenvolvimento de *software*. A definição inadequada de requisitos é responsável por uma parte significativa dos erros detectados ao longo do processo de desenvolvimento de sistemas, principalmente no caso de sistemas dedicados, de tempo real e crítico (LUTZ, 1993). A eliminação de erros de especificação torna-se cada vez mais difícil e dispendiosa à medida que o sistema avança para etapas posteriores de seu ciclo de vida, como projeto e implementação (DAVIS, 1993).

A definição de requisitos não é uma atividade desenvolvida de forma simples e direta. É uma atividade interativa, com a presença do elemento humano e de muita comunicação, sendo necessária a integração de pessoas, de ferramentas e de informações.

Neste capítulo, apresentamos conceitos sobre requisitos e engenharia de requisitos, o processo da engenharia de requisitos e as técnicas de modelagem baseadas em cenários.

2.1 CONCEITOS: REQUISITOS E ENGENHARIA DE REQUISITOS

Na literatura, existem inúmeras definições para o termo requisitos.

Uma definição simples para requisitos é dada por MACAULAY (1996). Segundo ele, requisito é, simplesmente, algo de que o cliente necessita.

Segundo JACKSON (1995), requisitos são fenômenos ou propriedades do domínio da aplicação que devem ser executados, normalmente, expressos em linguagem natural, diagrama informal ou outra notação apropriada ao entendimento do cliente e da equipe de desenvolvimento.

A IEEE (IEEE, 1997) define requisito como sendo:

1) uma condição ou uma capacidade de que o usuário necessita, para solucionar um problema ou alcançar um objetivo;

2) uma condição ou uma capacidade que deve ser alcançada ou possuída por um sistema ou componente do sistema, para satisfazer um contrato, um padrão, uma especificação ou outros documentos impostos formalmente;

3) uma representação documentada de uma condição ou capacidade, conforme os itens (1) e (2).

A definição da IEEE cobre a visão do usuário sobre requisitos (comportamento externo do sistema), a visão do desenvolvedor (2) e o conceito fundamental de que requisitos devem ser documentados (3).

Outra definição necessária é a de engenharia de requisitos. Uma definição genérica para o termo é uma atividade que objetiva estabelecer o que o cliente requer de um sistema de *software*. Podemos encontrar, na literatura, várias definições da engenharia de requisitos. A seguir, faremos uma revisão de algumas definições.

Segundo o IEEE (IEEE, 1984), engenharia de requisitos corresponde ao processo de aquisição, refinamento e verificação das necessidades do cliente para um sistema de *software*, objetivando-se ter uma especificação completa e correta dos requisitos de *software*.

Para LEITE (1987), a engenharia de requisitos pode ser definida como um processo segundo diferentes pontos de vista, no qual o "o que" é para fazer, é capturado e modelado. Nesse processo, são usados uma combinação de métodos, ferramentas e atores, sendo produzido, como resultado da modelagem, um documento de requisitos.

Segundo KOTONYA e SOMMERVILLE (1997), engenharia de requisitos trata-se de um termo usado para cobrir todas as atividades envolvidas na descoberta,

na documentação e na manutenção de um conjunto de requisitos, em um sistema baseado em computador. O termo engenharia implica que técnicas sistemáticas e repetitivas são usadas para assegurar que os requisitos do sistema sejam completos, consistentes e relevantes.

Para ZAVE (1997), a engenharia de requisitos é uma área ampla e multidisciplinar, onde aspectos sociais e humanos desempenham um importante papel no processo de engenharia de requisitos.

A definição de IEEE (1997) e de KOTONYA e SOMMERVILLE (1997) para requisitos e engenharia de requisitos, respectivamente, fundamentam a proposta do trabalho.

2.2 CLASSIFICAÇÃO DOS REQUISITOS

De forma geral, podem-se dividir os requisitos em funcionais e não-funcionais (KOTONYA; SOMMERVILLE, 1997).

Os requisitos funcionais dizem respeito à definição das funções que um sistema ou um componente de sistema deve possuir. Eles descrevem as transformações a serem realizadas nas entradas de um sistema ou em um de seus componentes, a fim de que se produzam saídas.

Os requisitos não-funcionais dizem respeito às restrições, aspectos de desempenho, interfaces com o usuário, confiabilidade, segurança, manutenibilidade, portabilidade, padrões, custos operacionais e outras propriedades que o sistema deve possuir, bem como aspectos sociais e políticos (CHUNG *et al.*, 2000). Alguns desses requisitos são, provavelmente, traduzidos em funções (operacionalizados), ao longo do processo de desenvolvimento de *software*. Os requisitos não-funcionais desempenham um papel crítico durante o desenvolvimento de sistemas, e erros devido à não-elicitação ou à elicitação incorreta desses estão entre os mais caros e difíceis de corrigir, uma vez que um sistema tenha sido implementado (CYSNEIROS, 1999).

De forma geral, a diferença entre requisitos funcionais e não-funcionais está no fato de os primeiros descreverem "o que" o sistema deve fazer, enquanto que os outros fixam restrições sobre "como" os requisitos funcionais serão implementados.

2.3 O PROCESSO DE ENGENHARIA DE REQUISITOS

O processo de engenharia de requisitos pode ser considerado como um conjunto estruturado de atividades que são seguidas com o objetivo de derivar, validar e manter um documento de requisito (KOTONYA; SOMMERVILLE, 1997). Para entender o processo de engenharia de requisitos, vejamos a Figura 1.

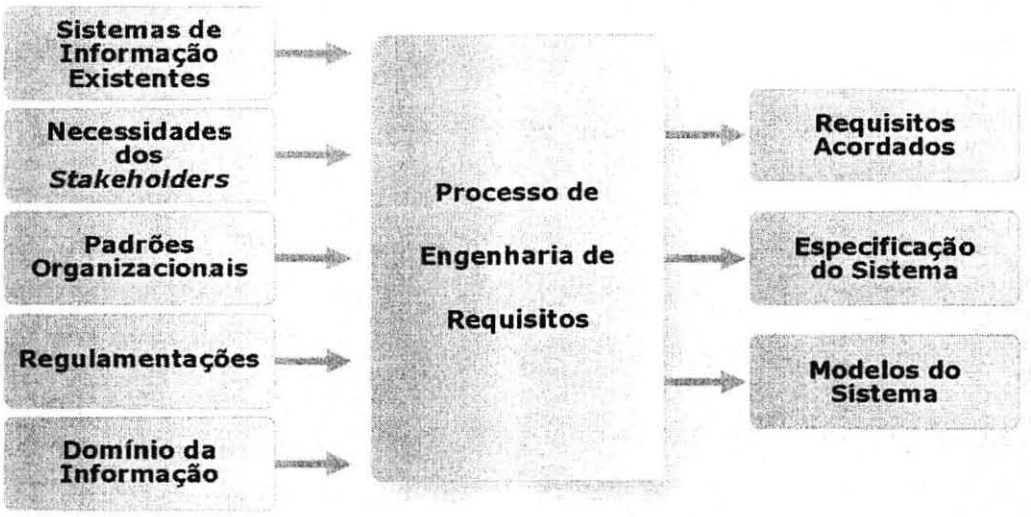


FIGURA 1 - ENTRADAS E SAÍDAS DO PROCESSO DE ENGENHARIA DE REQUISITOS (KOTONYA; SOMMERVILLE, 1997)

Verificamos que as entradas para o processo de engenharia de requisitos incluem: *informações sobre sistemas já existentes*, *necessidades dos stakeholders*, *padrões organizacionais*, *regulamentações* e *informações do domínio da aplicação*.

Todas essas informações são utilizadas para a realização das atividades do processo. Como resultado (saída), obtemos os *requisitos acordados, uma especificação de requisitos e modelos do sistema*. Essa visão macro ressalta que, para a realização das atividades do processo de engenharia de requisitos, fatores humanos e técnicos têm de ser adequadamente tratados, objetivando, dessa forma, que cada resultado do processo seja o mais completo e consistente possível.

Este trabalho adotará o modelo proposto por KOTONYA e SOMMERVILLE (1997), que descreve as seguintes fases do processo de requisitos: elicitação, análise e negociação, documentação e validação dos requisitos. A Figura 2 mostra essas fases representadas em um modelo espiral, no qual cada fase do processo é repetida até que seja tomada a decisão de que o documento de requisitos pode ser aceito. Além disso, as mudanças de requisitos são parte da fase de gerenciamento de requisitos. Apesar de essas atividades serem, normalmente, descritas independentemente e em uma ordem particular, na prática, elas consistem de processos iterativos e inter-relacionados que podem cobrir todo o ciclo de vida do desenvolvimento de sistemas de *software* (NUSEIBEH; EASTERBROOK, 2000).

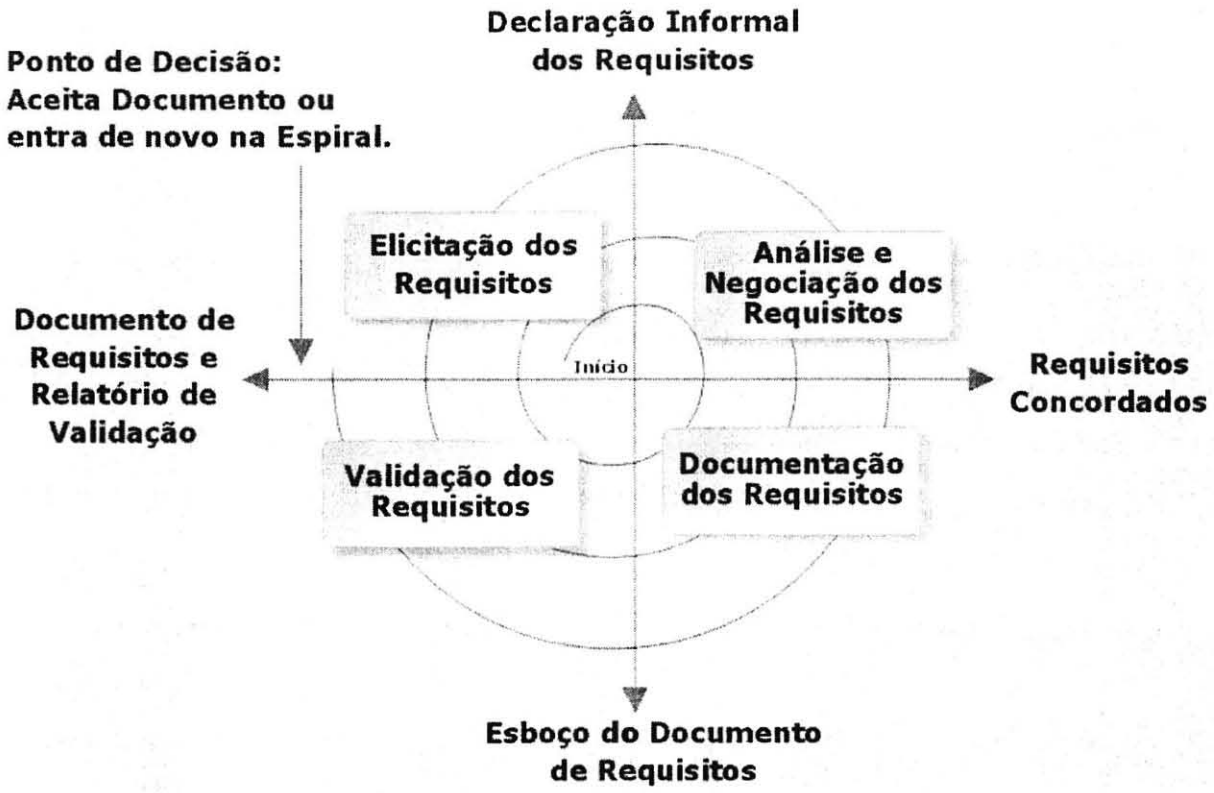


FIGURA 2 - PROCESSO DA ENGENHARIA DE REQUISITOS NO MODELO ESPIRAL

2.3.1 Elicitação dos Requisitos

Esta é a primeira fase no ciclo de vida da engenharia de requisitos, o seu propósito geral é obter conhecimento relevante para o problema a ser resolvido. A elicitação de requisitos é um processo de descoberta dos requisitos para um sistema

através da comunicação entre os *stakeholders*. Além de descobrir quais são as necessidades dos usuários, essa fase também requer uma cuidadosa análise da organização, do domínio de aplicação e dos processos organizacionais (KOTONYA; SOMMERVILLE, 1997). Assim, nesta fase, os desenvolvedores têm a tarefa de identificar os fatos que compõem os requisitos do sistema, de forma a prover o mais correto entendimento do que é esperado do sistema *software*. Outro papel da fase de elicitação é identificar as necessidades das diferentes classes de usuários, por exemplo: experientes, ocasionais, iniciantes, etc. (SHARP *et al.*, 1999).

Um dos principais problemas encontrados durante essa fase é a dificuldade de entender as reais necessidades dos usuários (NUSEIBEH; EASTERBROOK, 2000). Em boa parte dos casos, isso é devido à formação distinta entre analistas e usuários, gerando pontos de vista diferentes sobre o mesmo problema. Outro fator que dificulta o processo de elicitação é que, muitas vezes, os usuários não têm uma idéia precisa e explícita do sistema a ser desenvolvido, ou mesmo possuem dificuldades em descrever seu conhecimento sobre o domínio do problema.

A escolha da técnica apropriada para elicitar requisitos depende do tempo e dos recursos disponíveis, assim como do tipo de informação necessária. Algumas das classes de técnicas de elicitação são (NUSEIBEH; EASTERBROOK, 2000):

- técnicas tradicionais – incluem o uso de questionários, entrevistas, análise de documentação existente (MAMANI; LEITE, 1999);
- técnicas de elicitação em grupo – são técnicas de dinâmica de grupo com o objetivo de entender de forma mais detalhada as necessidades dos usuários, estão incluídas: *brainstorming* e sessões *Joint Application Design* (JAD) (DAMIAN, 2002);
- prototipação – é utilizada para elicitar requisitos quando há um alto grau de incerteza ou quando é necessário um rápido *feedback* dos usuários (DAVIS, 1992);
- técnicas de modelagem – fornecem um modelo específico das informações que serão adquiridas, e usam esse modelo para orientar o processo de elicitação. Incluem métodos baseados em metas, tais como: *KAOS*

(DARDENE *et al.*, 1993) e I* (YU, 1995) e métodos baseados em cenários (JACOBSON, 1995) (LEITE *et al.*, 1997) (ROLLAND *et al.*, 1998) (HAUMER *et al.*, 1998).

- técnicas cognitivas – incluem uma série de técnicas originalmente desenvolvidas para aquisição de conhecimento para sistemas baseados em conhecimento, alguns exemplos são: análise de protocolo, *laddering*, *card sorting*, *repertory grids* (SHAW; GAINES, 1996);
- técnicas contextuais – incluem técnicas de etnografia e análise social (VILLER; SOMMERVILLE, 1999), surgiram como uma alternativa para as técnicas tradicionais e cognitivas.

O resultado final da fase de elicitação é um esboço de documento que contém uma descrição dos requisitos.

2.3.2 Análise e Negociação dos Requisitos

As atividades da fase de análise e de negociação de requisitos são, na maioria das vezes, executadas de forma paralela ou intercalada. Durante o processo de análise, os requisitos são analisados, a fim de se detectarem incompletudes, omissões e redundâncias e, assim, descobrir os requisitos que realmente são necessários e que o usuário deseja. Dentre as técnicas que podem ser utilizadas no processo de análise, podemos citar: *checklist* de análise, matrizes de interação e prototipação.

O processo de negociação de requisitos tenta resolver conflitos entre usuários sem, necessariamente, comprometer a satisfação dos objetivos de cada usuário. Em geral, os modelos de negociação identificam as principais necessidades de cada usuário, ou seja, atribuem prioridades aos requisitos; em seguida, analisam esses resultados para tentar garantir que os requisitos mais críticos sejam atendidos.

2.3.3 Documentação dos Requisitos

Depois que os requisitos do sistema são aceitos, eles devem ser documentados com um nível apropriado de detalhes. Em geral, é necessário que o documento de requisitos possa ser entendido por todos os envolvidos no processo de engenharia de requisitos, pois ele servirá como um contrato entre usuários e desenvolvedores.

O principal foco da pesquisa em documentação de requisitos é prover notações e linguagens de especificação. Desde linguagem natural (AMBRIOLA; GERVASI, 1997) à lógica (CASTRO *et al.*, 1996) (ANTONIOU, 1998), diferentes linguagens têm sido propostas para expressar e descrever requisitos. Pesquisas atuais têm reconhecido que o gerenciamento de requisitos é uma atividade crucial no processo de engenharia de requisitos, ou seja, é necessário não somente escrever os requisitos de forma entendível, mas também permitir que eles possam ser rastreados e gerenciados ao longo da evolução do sistema (GOTEL; FINKELNSTEIN, 1994). Rastreamento de requisitos é um fator importante para prover uma documentação dos requisitos de forma completa e com integridade, assim como para ajudar no processo de gerenciamento de mudanças nesses requisitos (TORANZO; CASTRO, 1999).

Segundo RYAN (1993), o documento de requisitos é o meio através do qual é possível descrever as restrições quanto às características do produto e quanto ao processo de desenvolvimento, a interface com outras aplicações, a descrição sobre o domínio e as informações de suporte ao conhecimento do problema.

2.3.4 Validação dos Requisitos

Segundo LOUCOPOULUS e KARAKOSTAS (1995), a fase de validação de requisitos é definido como a atividade na qual se certifica que o documento de requisitos está consistente com as necessidades dos usuários.

Descrever os requisitos de forma explícita é uma condição necessária não somente para validar os requisitos, mas também para resolver conflitos entre usuários. Em geral, a validação de requisitos é considerada uma atividade complicada por dois

principais motivos. Inicialmente, pela própria natureza filosófica desse processo, no qual é levado em consideração o que é verdadeiro, permitindo que muitos pesquisadores comparem a validação de requisitos com o problema de validação do conhecimento científico (NUSEIBEH; EASTERBROOK, 2000). O segundo motivo é social e está relacionado com a dificuldade de obter um consenso entre diferentes usuários com objetivos conflitantes (LAMSWEERDE *et al.*, 1998). Outro grande desafio durante a validação de requisitos é demonstrar que a especificação dos requisitos do sistema está correta (KOTONYA; SOMMERVILLE, 1997). Contudo existem várias técnicas que podem ser aplicadas para suportar a validação de requisitos: revisões dos requisitos, prototipação, testes de requisitos e validação de modelos.

2.3.5 Gerenciamento dos Requisitos

Esta fase é uma das mais importantes do processo de engenharia de requisitos. Ela tem como objetivo principal o controle e o gerenciamento de mudanças nos requisitos (TORANZO; CASTRO , 1999).

De forma resumida, podemos descrever os objetivos do gerenciamento de requisitos como:

- gerenciar mudanças para os requisitos acordados;
- gerenciar o relacionamento entre requisitos;
- gerenciar as dependências entre o documento de requisitos e os demais documentos produzidos no processo de engenharia de requisitos.

Na fase de gerenciamento de requisitos, os requisitos devem ser identificados. Essa identificação é o meio adotado para poder rastrear e avaliar os impactos advindos de mudanças. Para grandes sistemas, nos quais o número de requisitos a serem gerenciados é muito grande, é necessário que os mesmos sejam armazenados em uma base de dados e sejam registradas as ligações entre os requisitos relacionados. Atualmente, verifica-se que, na grande maioria dos sistemas, faz-se necessário o uso

de ferramentas *CASE* para apoiar o processo de gerenciamento de requisitos. Grandes quantidades de requisitos não são gerenciáveis sem a utilização de alguma ferramenta computacional de apoio (WIEGERS, 1999). Algumas ferramentas existentes atualmente no mercado usadas para este fim são *RequisitePro* (*RATIONAL*, 2003), *DOORS* (TELELOGIC, 2003) e *CaliberRM* (BORLAND, 2003).

2.4 TÉCNICAS DE MODELAGEM BASEADAS EM CENÁRIOS

Na engenharia de requisitos, dada à dificuldade de elicitar, analisar, negociar e validar os requisitos de *software*, técnicas baseadas em cenários têm sido consideradas de grande utilidade. Isso porque cenários são construídos do ponto de vista de clientes/usuários, através de uma linguagem facilmente entendida pelos mesmos. As descrições do sistema realizadas através de cenários devem ser entendidas também pelos demais *stakeholders*. Assim, o trabalho para elicitar, analisar e validar os requisitos de um sistema pode integrar todos os *stakeholders*, num esforço em conjunto, que leva ao desenvolvimento de um documento de requisitos mais completo, consistente e não ambíguo.

Objetiva-se com o uso de cenários descrever as ações em um ambiente relacionadas a um sistema atual ou a um sistema a ser desenvolvido.

Algumas abordagens propondo a utilização de cenários para elicitação e validação de requisitos incluem: *USE CASE* (JACOBSON, 1995), a proposta apresentada em LEITE *et al.* (1997) e a proposta apresentada pelo projeto CREWS (*Cooperative Requirements Engineering With Scenarios*) (RALYTÉ, 1999) e as abordagens (HAUMER *et al.*, 1998) (ROLLAND *et al.*, 1998) (SUFTCLIFFE, 1998) (DUBOIS; HEYMANS, 1998).

Normalmente, a linguagem utilizada para essas descrições é a linguagem natural. No entanto, técnicas baseadas em cenários, tais como *USE CASE* (JACOBSON, 1995), também utilizam diagramas e notações gráficas. Propostas para representações formais podem ser encontradas em HSIA *et al.*, (1994) e em DUBOIS e HEYMANS (1998) e propostas totalmente informais podem ser encontradas em

KUUTTI (1995) e ROSSON e CARROLL (1995). Revisões extensas da utilização de cenários existem e estão disponíveis em FILLIPIDOU (1998), ROLLAND *et al.* (1998a) e WEIDENHAUPT (1998).

Os trabalhos na engenharia de requisitos relacionados com técnicas baseadas em cenários têm sido crescentes. Desde os anos 80, os resultados desses trabalhos vêm sendo divulgados em jornais e em revistas da área. Há alguns anos, *USE CASE* (JACOBSON, 1995) tem sido adotada como integrante do modelo padrão de linguagem de modelagem, a *UML* (*Unified Modeling Language*) (BOOCH *et al.*, 1999). Segundo SANTANDER (2002), essa é uma das provas mais fortes da relevância de técnicas baseadas em cenários. Assim, cada vez mais, reconhece-se, em cenários, uma forma efetiva de auxílio no entendimento e especificação de requisitos.

Nas subseções a seguir, são apresentadas algumas propostas para a representação de cenários.

2.4.1 Método para a Análise de Requisitos Baseado em Cenários (*SCRAM*)

Para BREITMAN (2000), neste enfoque é utilizada uma combinação das técnicas de prototipação, entrevistas, cenários e *rationale.* Com base na hipótese de que a integração de técnicas fornece o melhor caminho para a engenharia de requisitos, é proposto um método de integração em quatro estágios (SUTCLIFFE, 1998):

- entrevistas e técnicas para descobrimento de fatos – estas técnicas são utilizadas de modo a elicitar dados suficientes que permitam a construção de protótipo, chamado demonstrador de conceito;
- construção de protótipos – podem-se utilizar ferramentas comerciais, tais como o *Macromedia Director* e *Microsoft Visual Basic*;
- validação com clientes – os protótipos são utilizados para validar os requisitos junto aos clientes;

- análise – é proposto um *framework* específico para a análise dos requisitos. Nesta fase, é conduzida uma reunião do tipo *JAD*, com um cronograma previamente definido e um mediador externo para conduzir a sessão.

Segundo BREITMAN (2000), este método foi utilizado em vários estudos de caso. A combinação de técnicas provou ser útil na captura dos requisitos. O fato de utilizarem cenários na descrição de situações auxiliou em manter a atenção dos clientes. Por outro lado, o processo de desenvolvimento baseado em cenários provou ser fraco na captura de requisitos não-funcionais.

2.4.2 Ciclos de Questionamento (*Inquiry Cycle*)

Segundo BREITMAN (2000), neste enfoque é proposto um modelo para a descrição e o suporte de discussões sobre os requisitos do sistema que utiliza uma estratégia baseada em ciclos consecutivos de questionamento e refinamento destes requisitos. O método proposto é baseado em objetivos e os cenários são derivados a partir da identificação e decomposição destes. Uma vez identificados, os cenários farão parte de uma estratégia dinâmica que, segundo POTTS *et al.* (1994), permite o questionamento dos requisitos do sistema.

O ciclo se dá da seguinte forma: a documentação relativa aos requisitos, consiste dos cenários identificados e de uma lista de requisitos, que é discutida através de um processo de questionamento onde respostas e justificativas (*rationale*) para as questões são registradas. O processo de questionamento só termina quando uma decisão é tomada. As decisões resultantes desse processo podem implicar a mudança dos requisitos, o que, por sua vez, resulta na modificação da documentação e justifica um novo ciclo.

Cenários são utilizados na validação e no esclarecimento dos requisitos. A comparação, o refinamento e a avaliação de cenários são realizados ao nível dos objetivos. Os cenários podem ser documentados de modos diferentes, dependendo do nível de detalhe necessário. A forma mais simples é um *use case*, que consiste em uma

breve descrição acrescida de um número identificador. Formas mais detalhadas são chamadas de *scripts* e são representadas através de tabelas ou de diagramas. Segundo BREITMAN (2000), esse modelo, apesar de pregar a utilização de cenários, não aponta como e onde os mesmos devem ser elicitados.

2.4.3 Cenários como Apoio à Visualização de Requisitos

ZORMAN (1995), *apud* BREITMAN (2000), discute a dificuldade na comunicação entre clientes e desenvolvedores, apontando ocorrências de mistura de terminologia e erros de interpretação. Os especialistas de domínio e *software* necessitam colaborar com uma representação e vocabulários comuns para a elaboração de cenários. Sob esse aspecto, cenários podem vir a ser utilizados eficientemente na comunicação entre pessoas com *backgrounds* diferentes. Linguagens mais formais devem ser utilizadas na comunicação com pessoas de uma mesma área.

ZORMAN (1995) propõe a utilização de uma estratégia para a visualização (*envisaging*) dos requisitos do sistema. Esse processo, que conta com o auxílio de cenários, consiste na transformação de noções informais do que se deseja de um sistema em uma descrição mais precisa do mesmo, traduzida pelo documento de especificação.

Cenários, nesse enfoque, são definidos como descrições parciais do sistema e do comportamento do macrosistema. ZORMAN (1995) apresenta uma ferramenta para a captura e a representação de cenários, *REBUS,* que se assemelha à representação utilizada por *storyboards*. Cada cenário é composto de um nome, categoria, descrição e uma representação gráfica.

Segundo ZORMAN (1995), a maior contribuição de seu trabalho reside no aumento da qualidade da comunicação entre clientes e desenvolvedores através da utilização da técnica de cenários. A representação para cenários proposta é independente de domínio de aplicação dos mesmos. Essa assertiva é justificada no fato de que a representação para cenários foi elaborada utilizando-se blocos de construções derivados de teorias lingüísticas.

2.4.4 Utilização de Cenários para Elicitar Objetivos

Esta abordagem utiliza um enfoque baseado em objetivos para a identificação dos requisitos do sistema. Dentro desse contexto, propõe-se a utilização da técnica de cenários para auxiliar a elicitação desses objetivos. Segundo ROLLAND *et al.* (1998a), um cenário pode ser definido como "um comportamento possível limitado a um conjunto de interações propositais que são levadas a cabo por uma série de agentes".

A notação proposta para cenários é composta de uma ou mais ações, em que a combinação dessas descreve um caminho único que leva os agentes de um estado inicial a um estado final. Dessa forma, a combinação de vários cenários é capaz de descrever as interações entre um sistema complexo de agentes (BREITMAN, 2000).

Essa notação para cenários utiliza um formato textual semi-estruturado e linguagem natural.

2.4.5 *Use Cases* (Casos de Uso)

Use cases em *UML* (BOOCH *et al.*, 1999) são utilizados para descrever casos de uso de um sistema por atores. Um ator representa qualquer elemento externo que interage com o sistema. Um *use case* descreve uma seqüência de passos/operações que um usuário realiza quando interage com um sistema visando realizar uma determinada tarefa/objetivo. Assim, o aspecto comportamental de um sistema a ser desenvolvido pode ser descrito. No entanto, a descrição de *use cases* não trata a questão de como essa interação será implementada. Fases posteriores à etapa de engenharia de requisitos, tais como projeto e implementação, focalizarão esse aspecto.

Segundo SANTANDER (2002), um *use case* pode gerar vários cenários. Cenários estão para *use cases* assim como instâncias estão para classes, significando que um cenário é basicamente uma instância de um *use case*. Um *use case* envolve uma situação de utilização do sistema por um ator. Nessa situação, vários caminhos podem ser seguidos dependendo do contexto na execução do sistema. Esses caminhos

são os possíveis cenários do *use case*. O caminho básico para realizar um *use case*, sem problemas e sem erros em nenhum dos passos da seqüência, é denominado de cenário primário. Nesse tipo de cenário, a execução dos passos para realizar a funcionalidade básica do *use case* é obtida com sucesso. Por outro lado, caminhos alternativos, bem como situações de erro, podem ser representados através de cenários secundários. Cenários secundários podem ser descritos separadamente ou como extensão da descrição de um cenário primário.

Outras técnicas também podem ser usadas em *UML* para refinar fluxos de eventos em *use cases*. A idéia consiste basicamente em incluir relacionamentos que permitam descrever diversos aspectos de comportamento entre *use case*. Segundo SANTANDER (2002), os relacionamentos apontados em *UML* incluem:

- relacionamento do tipo ***<<include>>***: quando for detectado, no sistema, um conjunto de passos comuns a vários *use case*; pode-se criar um *use case* com esses passos, com potencial para ser reutilizado por outros *use case*;
- relacionamento do tipo ***<<extend>>***: é utilizado quando existe uma seqüência opcional ou condicional de passos que queremos incluir em um *use case*;
- relacionamento do tipo ***<<generalization>>***: generalização entre *use case* tem o mesmo significado de generalização entre classes na orientação a objetos. Isso significa que um *use case* "filho" herda o comportamento e a estrutura do *use case* "pai". Considera-se que um *use case* "filho" é uma especialização do *use case* "pai", podendo adicionar nova estrutura e comportamento bem como modificar o comportamento do caso de uso "pai".

Da mesma forma que se permite o uso do mecanismo de generalização entre *use cases*, pode-se usar o relacionamento de generalização entre atores representados em diagrama de *use case*.

A Figura 3, apresenta as notações básicas utilizadas para descrever *use case*.

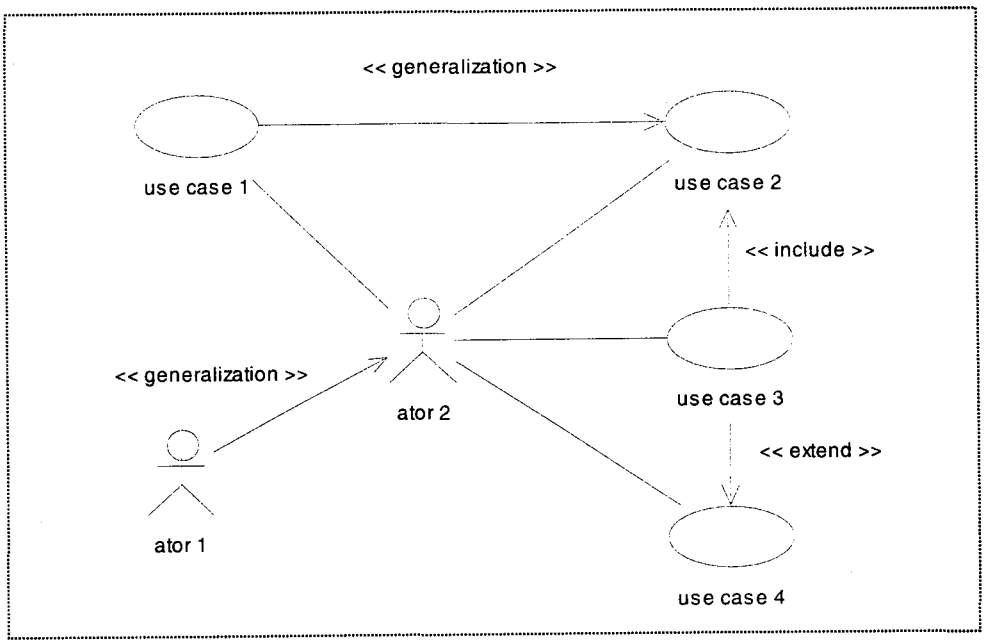


FIGURA 3 - NOTAÇÕES PARA *USE CASE* EM *UML* (SANTANDER, 2002)

Na figura acima, o ator 2, por exemplo, pode ser uma especialização do ator 1. Nesse caso, o ator 2 herda toda a estrutura e o comportamento do ator 1 e pode adicionar nova estrutura e comportamento em relação ao ator 1. Outras técnicas adicionais propostas em *UML* para representar e descrever melhor os *use case* podem ser vistas em BOOCH *et al.*, (1999).

Segundo SANTANDER (2002), é consensual que *use cases* não são suficientes para detalhar todos os elementos que devem ser definidos no processo de engenharia de requisitos. No entanto, as vantagens do uso dessa técnica, como também de outras técnicas baseadas em cenários, são que podemos, juntamente com as descrições de interações entre um usuário e o sistema, relacionar outros tipos de requisitos, tais como requisitos não-funcionais e organizacionais bem, como evoluir posteriormente para outros artefatos no processo de desenvolvimento.

Como exemplo, podemos citar o Processo Unificado (*Unified Process*) (JACOBSON *et al.*, 1999), o qual adota a descrição de *use case* como fonte de informações para gerar: diagramas de classes, diagramas de seqüência, bem como descrições arquiteturais do *software.*

2.4.6 Cenários no Contexto da *Baseline*[1] de Requisitos

LEITE *et al.* (1997) definem cenários como descrições evolutivas de situações próprias ao ambiente. Esse enfoque prega que cenários devem ser utilizados durante todo o processo de desenvolvimento do *software*. Compreende, além da interação entre sistema e clientes, a interação entre módulos do sistema. Assim, temos desde cenários iniciais, que modelam o macrosistema onde o sistema será desenvolvido e instalado, até cenários externos, isto é, que representam as interações dos usuários com o sistema quando este estiver em funcionamento, passando por várias versões intermediárias ao longo do processo de desenvolvimento do *software*.

Os pontos centrais da abordagem proposta por LEITE *et al.* (1997) são:

- cenários descrevem situações que ocorrem no macrosistema e suas relações com o sistema;
- cenários evoluem durante o processo de desenvolvimento de *software*;
- cenários estão, naturalmente, ligados ao *Language Extended Lexicon* (*LEL*) do Universo de Informações do sistema (*UdI*);
- cenários são descritos em linguagem natural.

A estrutura de um cenário, proposta por LEITE *et al.* (1997), está representada na Figura 4.

[1] A *baseline* de requisitos é definida como uma estrutura perene que incorpora artefatos de *software*. Ela é desenvolvida durante a fase de requisitos e evolui ao longo da vida útil do *software*. A *baseline* é paralela ao eixo de desenvolvimento de *software* e trata da evolução de todos os artefatos produzidos durante esse processo (LEITE *et al.*, 1997).

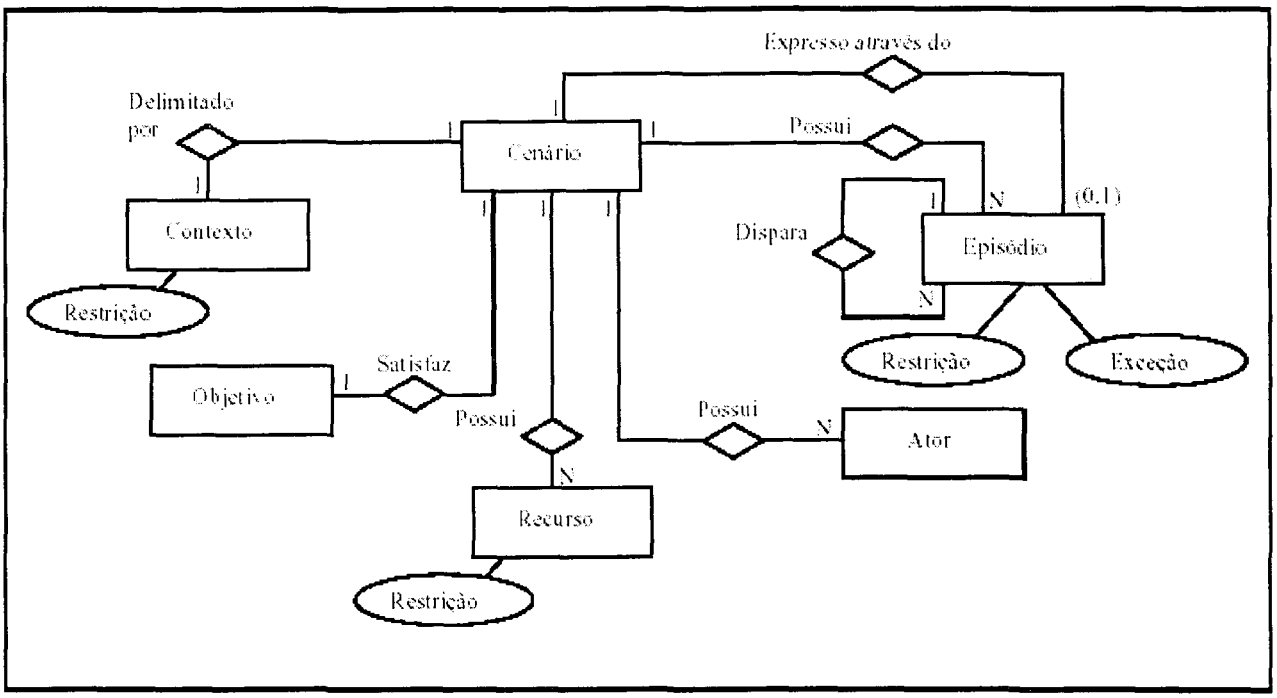


FIGURA 4 - DIAGRAMA ENTIDADE-RELACIONAMENTO DESCREVENDO CENÁRIOS (LEITE *et al.*, 1997)

São componentes dos cenários (LEITE *et al.*, 1997):

- título (*title*): identifica o cenário;
- objetivo (*goal*): estabelece a finalidade de um cenário. O cenário deve descrever o modo como esse objetivo deve ser alcançado;
- contexto (*context*): descreve o estado inicial de um cenário, suas pré-condições, o local (físico) e o tempo;
- recursos (*resources*): identifica os objetos passivos com os quais lidam os atores;
- atores (*actors*): pessoa ou estrutura organizacional que tem um papel no cenário;
- episódios (*episodes*): cada episódio representa uma ação realizada por um ator, onde participam outros atores utilizando recursos disponíveis. Um episódio também pode referir-se a outro cenário. Episódios podem conter restrições (*constraint*) e/ou exceções (*exception*). Uma restrição é qualquer imposição que restrinja um episódio de um cenário. Uma exceção é o tratamento para uma situação excepcional ou de erro.

A Figura 5, ilustra o processo de construção de cenários. As atividades do processo de construção são: derivar, descrever, organizar, verificar e validar cenários. A idéia geral do processo é partir do *LEL* e construir a primeira versão dos cenários. Esses cenários são então completados e organizados com a finalidade de obter um conjunto de cenários que representem o *UdI*. Durante ou depois das atividades, os cenários são verificados e validados com os clientes/usuários. Observa-se que essas atividades não ocorrem de forma seqüencial.

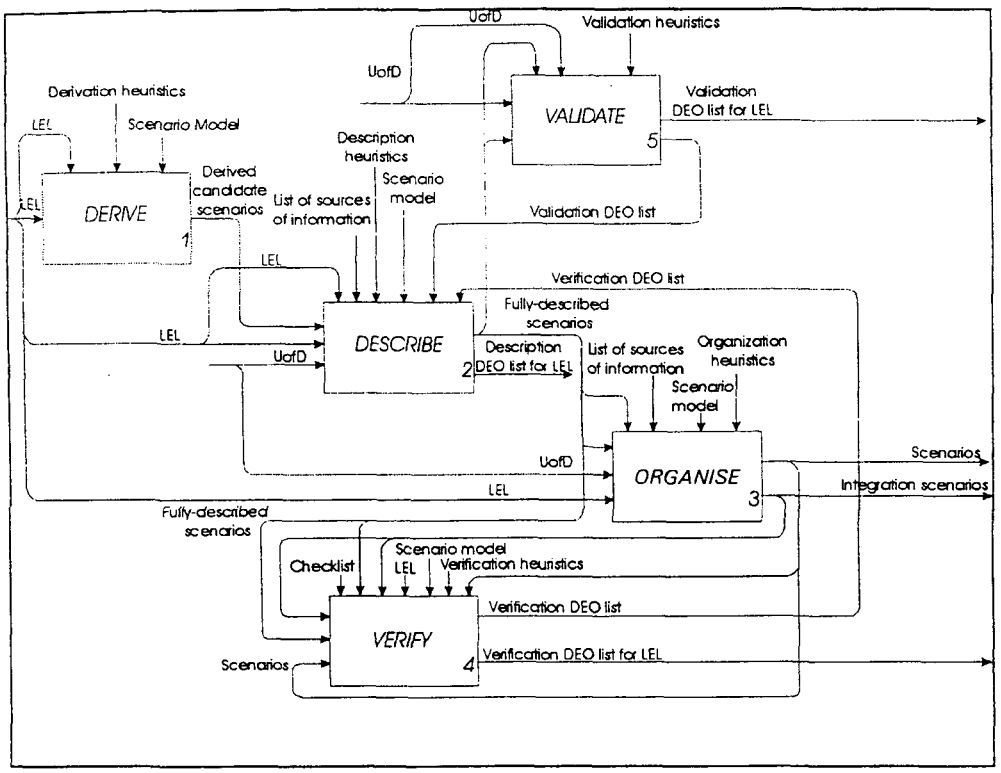


FIGURA 5 – O PROCESSO DE CONSTRUÇÃO DE CENÁRIOS (LEITE *et al.*, 2000)

Como dissemos anteriormente, nesta seção, cenários estão, naturalmente, ligados ao *LEL* do *UdI* do sistema. O principal objetivo do *LEL* é registrar a linguagem utilizada pelos atores do UdI, sem contudo se preocupar com a funcionalidade (LEITE; FRANCO, 1993). O *LEL* do *UdI* é composto por entradas, onde cada entrada está associada a um símbolo (palavra ou frase) da linguagem do *UdI*.

Cada símbolo pode possuir sinônimos e é descrito através de noções (*notion*) e impactos (*behavioral response*). As noções descrevem o significado e as relações

fundamentais de existência do símbolo com outros símbolos (denotação). Os impactos descrevem os efeitos do uso ou ocorrência do símbolo no *UdI* ou os efeitos de outras ocorrências de símbolos no *UdI* sobre o símbolo (conotação). Dependendo do símbolo que descrevem, as entradas podem ser classificadas como sujeito, verbo, objeto e estado (predicativo).

Ao se descreverem entradas do *LEL*, deve-se obedecer aos princípios de vocabulário mínimo e de circularidade. O princípio de vocabulário mínimo demanda que a utilização de símbolos externos ao *LEL* do *UdI* seja minimizada ao máximo. O princípio de circularidade implica a maximização da utilização de símbolos do *LEL* do *UdI* na descrição de símbolos. Os símbolos do *LEL*, que aparecem na descrição de símbolos, devem ser sublinhados. Como decorrência do princípio de circularidade, a forma natural de representação do *LEL* é pela utilização de uma estrutura de hipertexto. As entradas do *LEL* são os nós do hipertexto, enquanto os símbolos que aparecem nas descrições de símbolos são os elos do hipertexto. A Figura 6 apresenta uma entrada no *LEL* para um sistema de Controle de Eventos Científicos, utilizado como estudo de caso no capítulo 5. As palavras sublinhadas são elos para outros símbolos do *LEL*.

Crachá

Noção: Identificação do participante no evento.

Impacto:
- Emitido a todo participante.
- Necessário para acesso ao evento.
- Retirado na recepção do evento.

FIGURA 6 – EXEMPLO DE *LEL*: CRACHÁ

O processo de construção de *LEL*, Figura 7, engloba as atividades: identificar, classificar e descrever símbolos.

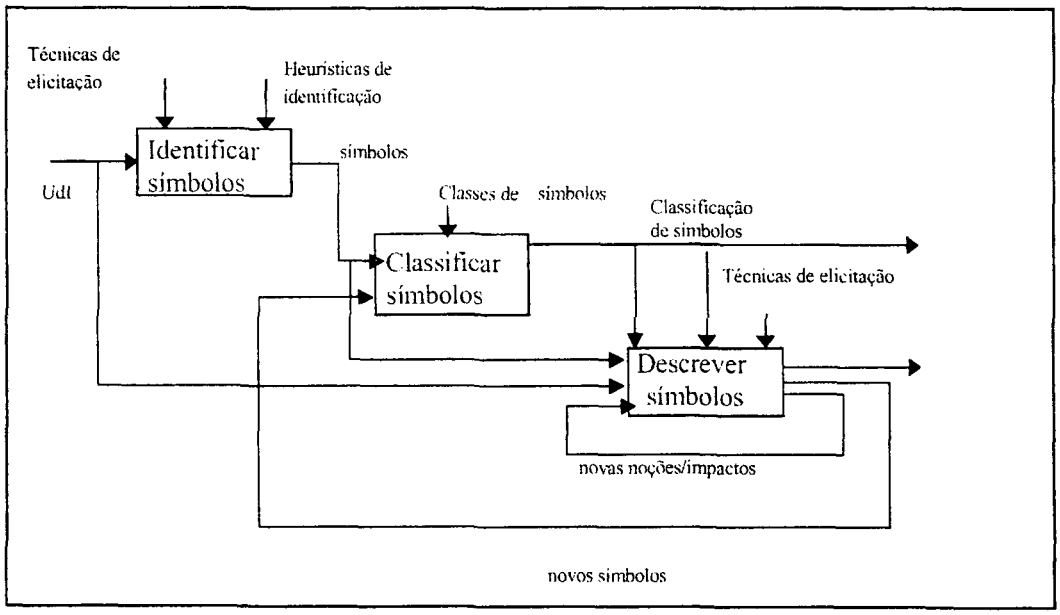


FIGURA 7 – PROCESSO DE CONSTRUÇÃO DO *LEL* (NETO, 2001)

Conforme mostrado na Figura 7, o processo de construção do *LEL* é continuado, sendo retomado sempre que as alterações surgirem no *UdI*, trazendo novos símbolos para o domínio ou pela descoberta de novos símbolos durante a elicitação de um dado símbolo (princípio da circularidade).

2.5 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentados os principais conceitos e técnicas de engenharia de requisitos. O nosso trabalho visa dar sua contribuição através da implementação de uma ferramenta que possibilita o uso de técnicas de modelagem de requisitos baseada em cenários (LEITE *et al.*, 1997) e (JACOBSON, 1995) para apoiar a *MDSODI* no ambiente *DiSEN*.

No próximo capítulo, descrevemos a *MDSODI* e o ambiente *DiSEN*.

3 DESENVOLVIMENTO DE *SOFTWARE* DISTRIBUÍDO

A complexidade dos atuais sistemas de *software* não pode mais ser enfrentada com as tradicionais técnicas e ferramentas de desenvolvimento. Torna-se necessário contar com o apoio de ferramentas que permitam, de uma maneira integrada, conduzir os projetos de forma rápida, controlada e com reduzido custo.

Visando suprir a necessidade de ferramentas e ambientes de desenvolvimento de *software* distribuído, foi criado, em 1999, o projeto de pesquisa "Metodologia para Desenvolvimento de Software Baseada em Objetos Distribuídos Inteligentes" (HUZITA, 1999), que está sendo desenvolvido no Laboratório de Engenharia de *Software* (LES) do Departamento de Informática (DIN) da Universidade Estadual de Maringá (UEM). Seu principal objetivo é definir uma metodologia de desenvolvimento de *software* baseado em objetos distribuídos bem como a construção de um ambiente integrado para o desenvolvimento de *software* distribuído no qual à metodologia proposta está inserida.

Para a execução desse projeto, têm sido definidos vários trabalhos, dentre os quais podemos citar: GRAVENA (2000), SILVA (2001), PASCUTTI (2002), PEDRAS (2003), YANAGA (2003), MORO (2003) e também este trabalho.

Este capítulo tem por objetivo descrever a *MDSODI* (Metodologia para Desenvolvimento de *Software* Distribuído) (GRAVENA, 2000) e apresentar o ambiente de desenvolvimento de *software* (*ADS*) distribuído denominado *DiSEN* (*Distributed Software Engineering Environment*), desenvolvido em PASCUTTI (2002).

3.1 METODOLOGIA PARA DESENVOLVIMENTO DE *SOFTWARE* DISTRIBUÍDO (*MDSODI*)

A *MDSODI* é uma Metodologia para Desenvolvimento de *Software* Distribuído proposta em GRAVENA (2000).

A metodologia *MDSODI* considera os aspectos de concorrência/paralelismo, distribuição, sincronização e comunicação em todas as fases do desenvolvimento do *software*. Esses aspectos são representados através da extensão da *UML* (GRAVENA, 2000) (HUZITA, 1995). Mantém as principais características do *Unified Process* (JACOBSON, 1999): dirigida a *use case*, centrada na arquitetura e no desenvolvimento iterativo e incremental. Utiliza também algumas notações propostas na Metodologia Orientada a Objetos para Desenvolvimento de *Software* para Processamento Paralelo (*MOOPP*) (HUZITA, 1995), como, por exemplo, a representação gráfica para os possíveis tipos de objetos e classes, tendo como base o paralelismo. Revisão extensa sobre a *MDSODI* pode ser encontrada em GRAVENA (2000).

O modelo do ciclo de vida da *MDSODI* pode ser visto na Figura 8 e as fases de cada incremento na Figura 9.

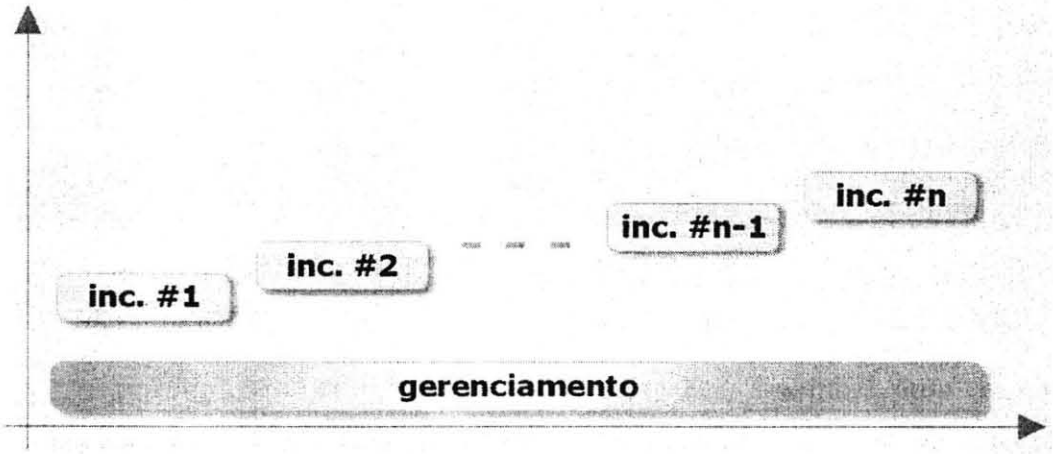


FIGURA 8 - CICLO DE VIDA DA *MDSODI*

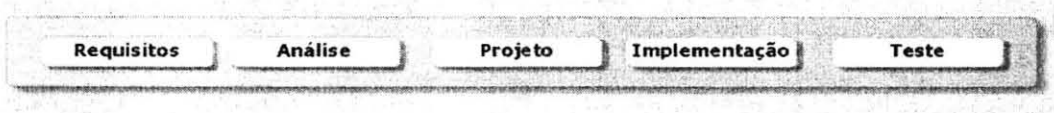


FIGURA 9 - FASES DE CADA INCREMENTO DO CICLO DE VIDA DA *MDSODI*

O processo de desenvolvimento de *software* da *MDSODI* ocorre seguindo as fases de requisitos, análise, projeto, implementação e testes (GRAVENA, 2000).

Requisitos: esta fase tem como objetivo principal identificar as funcionalidades necessárias para o desenvolvimento do sistema de uma forma adequada e eficiente a partir das necessidades do usuário. Nesta fase, devem-se entender os requisitos funcionais e não-funcionais e elaborar o diagrama de *use case* considerando os aspectos de paralelismo/concorrência e distribuição. Os artefatos produzidos são: modelo de negócio, diagrama de *use case* e descrição de requisitos.

Análise: com base nos requisitos identificados na fase anterior, assim como no diagrama de *use case* elaborado, definem-se as classes e os objetos do sistema, identificando aspectos de concorrência/paralelismo, distribuição e comunicação entre as classes. Devem-se também identificar os aspectos de concorrência/paralelismo e a distribuição entre pacotes através de descrição textual, quando necessário, e elaborar o diagrama de pacotes considerando esses aspectos. Nesta fase, são produzidos os artefatos: diagrama de classe, diagrama de colaboração, diagrama de estado, diagrama de pacotes e descrição arquitetural do modelo de análise.

Projeto: são objetivos desta fase elaboração do diagrama de classes de projeto, definição de aspectos da arquitetura do sistema, identificando a alocação dos pacotes nas camadas, e detalhamento de algoritmos, todos considerando os aspectos de paralelismo e distribuição. Nesta fase, são produzidos os artefatos: diagrama de seqüência (comunicação, paralelismo e sincronização), lista com os requisitos de implementação (linguagem e ambiente de programação escolhidos), localização física e questões de concorrência (entre métodos), diagrama de subsistema (considerando paralelismo, distribuição, sincronização e comunicação), divisão do sistema em camadas, aspectos arquiteturais do projeto, averiguação de componentes disponíveis para reutilização e detalhamento dos métodos.

Implementação: esta fase tem como objetivo principal a construção/implementação do sistema, tendo como base aspectos identificados nas fases de requisitos, análise e projeto. Nesta fase, devem ser definidas as interfaces entre os subsistemas identificados na fase de projeto. Deve-se também detalhar e implementar os métodos das classes já identificadas anteriormente. Os mecanismos de sincronização e os que tratam do balanceamento de carga entre os nós do

processamento, considerando os requisitos de distribuição, paralelismo, sincronização e comunicação, devem ser tratados.

Teste: constitui-se em um trabalho a ser desenvolvido (GRAVENA,2000).

Além das fases de um processo de desenvolvimento de *software*, a *MDSODI* propõe também a definição de uma notação adequada para a especificação do mesmo. O Quadro 1 ilustra os tipos de *use cases* existentes com suas respectivas notações.

QUADRO 1 - TIPOS DE *USE CASES* (GRAVENA, 2000)

Tipos de *use cases*	Notação
Use cases seqüenciais: *use cases* que agrupam um conjunto de funcionalidades que devem ser executadas seqüencialmente.	
Use cases paralelos: *use cases* que agrupam um conjunto de funcionalidades que devem ser executadas em paralelo.	
Use cases distribuídos: *use cases* que podem estar em diferentes locais no sistema.	
Use cases paralelos e distribuídos: podem estar em diferentes locais do sistema e devem ser executados em paralelo.	

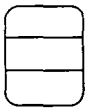
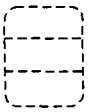
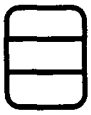
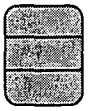
A *MDSODI* propõe também tipos de atores necessários para tratar os aspectos de sistemas distribuídos. O Quadro 2 mostra os tipos de atores e sua notação gráfica.

QUADRO 2 - TIPOS DE ATORES (GRAVENA, 2000)

Tipos de atores	Notação
Atores exclusivos: atores envolvidos com *use cases* seqüenciais.	
Atores paralelos: atores envolvidos em *use cases* paralelos.	
Atores distribuídos: atores que se encontram localizados em diferentes nós no sistema.	
Atores paralelos e distribuídos: atores que são, ao mesmo tempo, paralelos e distribuídos.	

A *MDSODI* considera também tipos de classes e objetos para tratar das características dos sistemas distribuídos. O Quadro 3 apresenta esses tipos de classes/objetos bem como sua representação gráfica.

QUADRO 3 - TIPOS DE CLASSES/OBJETOS (GRAVENA, 2000)

Tipos de classes/objetos	Notação
Classes e Objetos exclusivos: todos os seus métodos são executados seqüencialmente.	
Classes e Objetos parcialmente paralelos: alguns métodos são executados seqüencialmente enquanto outros, concorrentemente.	
Classes e Objetos totalmente paralelos: todos os seus métodos são executados concorrentemente.	
Classes e Objetos distribuídos: os métodos podem ser executados em diferentes nós no sistema.	

O Quadro 4, apresentado a seguir, mostra os tipos de relacionamentos. Não estão apresentados os relacionamentos convencionais da orientação a objetos, como, por exemplo, agregação e generalização. Porém os mesmos também poderão ser utilizados.

QUADRO 4 – TIPOS DE RELACIONAMENTOS (GRAVENA, 2000)

Tipos de relacionamentos	Notação
Relacionamento entre *use cases* exclusivos: *use cases* que são executados seqüencialmente. (Requisitos)	———
Relacionamento entre *use cases* paralelos: *use cases* que são executados concorrentemente com outros *use cases*. (Requisitos)	———
Relacionamento entre pacotes exclusivos: pacotes que são executados seqüencialmente. (Análise)	———
Relacionamento entre pacotes parcialmente paralelos: pacotes que têm algumas classes executadas de forma concorrente com outras classes de outro pacote.(Análise)	- - - - - - - - - - -

Relacionamento entre pacotes totalmente paralelos: pacotes em que todas as classes são executadas de forma concorrente com classes de outro pacote. (Análise)	
Relacionamento entre pacotes distribuídos: pacotes localizados em diferentes nós do sistema. (Análise)	
Relacionamento entre módulos/subsistemas exclusivos: subsistemas/Módulos que são executados seqüencialmente. (Projeto)	
Relacionamento entre módulos/subsistemas parcialmente paralelos: subsistemas/módulos que têm algumas classes executadas de forma concorrente com outras classes de outro subsistema. (Projeto)	
Relacionamento entre módulos/subsistemas totalmente paralelos: subsistemas/módulos em que todas as classes são executadas de forma concorrente com classes de outro subsistema.(Análise)	
Relacionamento entre subsistemas/módulos distribuídos: subsistemas/módulos localizados em diferentes nós dentro do sistema. (Projeto)	
Relacionamento entre objetos exclusivos: objetos que têm seus métodos executados seqüencialmente.	
Relacionamento entre objetos parcialmente paralelos: objetos que têm alguns métodos executados concorrentemente.	
Relacionamento entre objetos totalmente paralelos: objetos que têm todos os seus métodos executados concorrentemente.	
Relacionamento entre objetos distribuídos: objetos que se encontram localizados em diferentes nós do sistema.	

3.2 *DISTRIBUTED SOFTWARE ENGINEERING ENVIRONMENT (DISEN)*

Segundo PASCUTTI (2002), *DiSEN* é um ambiente distribuído de desenvolvimento de *software*, no qual a *MDSODI* está inserida, que tem como um de seus objetivos permitir que vários desenvolvedores, atuando em locais distintos, possam trabalhar de forma cooperativa no desenvolvimento de *software*.

3.2.1 A Arquitetura *DiSEN*

O estilo arquitetural do *DiSEN* é baseado em camadas. A arquitetura do *DiSEN* possui uma camada dinâmica que é a responsável pelo gerenciamento da (re)configuração do ambiente em tempo de execução; uma camada de aplicação que

tem como um dos elementos constituintes o suporte à *MDSODI*, e a camada da infra-estrutura que provê suporte às tarefas de persistência, nomeação e concorrência e, além disso, incorpora o canal de comunicação. Abaixo dessa camada, existe uma camada de *software* básico do sistema que irá conter, por exemplo, interfaces para *hardware* específico, sistema operacional, memória compartilhada e comunicação.

O canal de comunicação é um elemento fundamental da arquitetura, já que toda a comunicação entre os elementos constituintes da camada de aplicação e da camada da infra-estrutura será realizada através desse canal. É constituído pelo *middleware* e pelo *middle-agent*, sendo que, quando a comunicação envolver somente objetos, esta será feita pelo *middleware* e, quando os agentes estiverem envolvidos, devido às suas propriedades, a comunicação deverá ser gerenciada pelo *middle-agent*.

A Figura 10, mostra a representação gráfica da arquitetura do *DiSEN*.

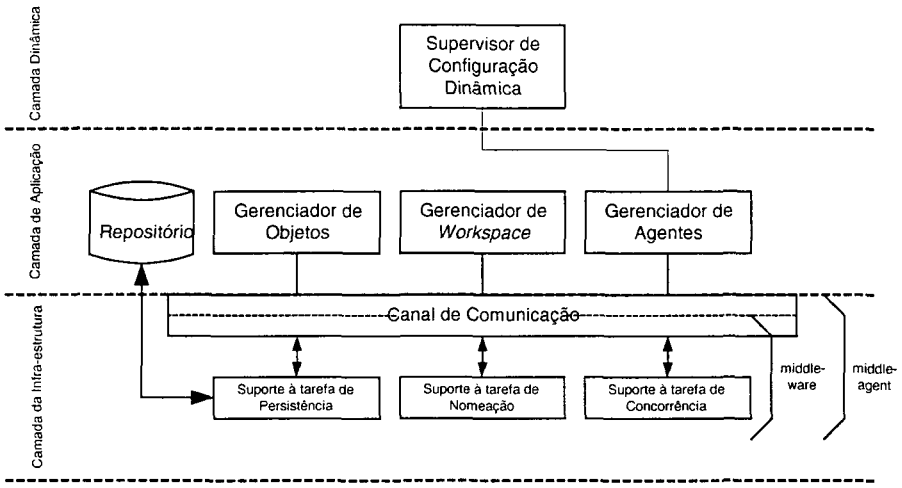


FIGURA 10 - ARQUITETURA DO *DISEN* (PASCUTTI, 2002)

Nas subseções a seguir, descrevemos os elementos da arquitetura *DiSEN*.

3.2.1.1 Supervisor de Configuração Dinâmica

Responsável pelo controle e gerenciamento da configuração do ambiente, bem como dos serviços que podem ser acrescentados ao ambiente em tempo de execução.

3.2.1.2 Gerenciador de Objetos

Responsável pelo controle e gerenciamento do ciclo de vida dos artefatos. Um artefato pode ser um diagrama, modelo, manual, código fonte ou código objeto. Esses artefatos são, por natureza, mais complexos que os itens tratados por sistemas de banco de dados tradicionais, pois são mais estruturados e requerem operações complexas.

O Gerenciador de Objetos é constituído por gerenciador de acesso, gerenciador de atividades, gerenciador de recursos, gerenciador de artefatos, gerenciador de projetos, gerenciador de processos e gerenciador de versões e configurações.

A seguir, descrevemos o gerenciador de recursos por ser de interesse para o presente trabalho; mais detalhes podem ser encontrados em PASCUTTI (2002).

Gerenciador de recursos: responsável pelo controle e gerenciamento de recursos para a execução de uma atividade. Os recursos necessários para a realização das atividades podem ser materiais (computador, impressora, sala de reunião) ou ferramentas (programas de computador destinados ao suporte ou automação de parte do trabalho relacionado com uma atividade). Para o ambiente *DiSEN,* a ferramenta *REQUISITE* é considerada como um de seus recursos.

3.2.1.3 Gerenciador de *Workspace*

O gerenciador de workspace é responsável pelo controle e gerenciamento da edição cooperativa de documentos e itens de *software* e também por administrar um conjunto de *workspaces*.

Segundo PASCUTTI (2002), o gerenciador de *workspace* provê funcionalidades para criar um ou mais *workspace*; isso será feito através do canal de comunicação e do suporte à tarefa de persistência, conforme mostra a Figura 11. No caso mais simples, isso consiste em copiar um arquivo simples, porém mais freqüentemente uma configuração inteira é copiada do repositório para o *workspace,* possibilitando ao desenvolvedor ter todos os módulos e documentos necessários para o sistema à sua disposição localmente. Assim, o desenvolvedor não tem de decidir o que

deverá ser mantido globalmente no repositório e o que é necessário localmente para carregar suas mudanças. Além disso, ele está isolado das alterações das outras pessoas para o repositório – e outras pessoas estão isoladas das suas alterações. Isso significa que ele tem um controle completo do seu mundo e sabe exatamente o que foi alterado e por quê.

Quando o desenvolvedor termina de efetuar suas modificações, ele precisa adicionar os módulos e documentos modificados no repositório. Essa operação, no caso mais simples, consiste em adicioná-las ao repositório, usando a funcionalidade do gerenciador de controle de versão. Entretanto, quando ele tem uma configuração completa no seu *workspace*, há provavelmente alguns arquivos que permaneceram inalterados e, por essa razão, não é preciso que sejam adicionados ao repositório. O gerenciador de *workspace* deve ser capaz de descobrir quais arquivos foram alterados e ter certeza de que todos esses – e somente esses – são adicionados no repositório. Isso poderia ser feito através da sincronização dos *workspaces* com o repositório, porém o trabalho de PASCUTTI (2002) não trata detalhes de como isso será realizado, constituindo-se, assim, em uma proposta a ser desenvolvida.

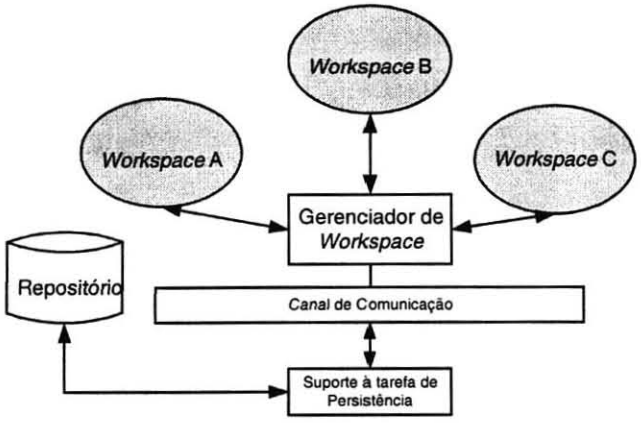


FIGURA 11 – REPRESENTAÇÃO DO GERENCIADOR DE *WORKSPACE* (PASCUTTI, 2002)

Em cada *workspace*, PASCUTTI (2002) define que haverá agentes locais que auxiliarão durante a execução das atividades, agentes de interação que auxiliarão nos trabalhos cooperativos e agentes de sistema que auxiliarão na coleta de métricas do sistema. Esses agentes interagirão com os demais, enviando mensagens através do canal de comunicação.

3.2.1.4 Repositório / Suporte à Persistência

Responsável pelo armazenamento dos artefatos, pelos dados das aplicações geradas pelo ADS, bem como pelo conhecimento necessário para a comunicação entre os agentes.

Para a persistência de artefatos, o *DiSEN* utiliza o repositório de metadados *MDR* (*Metadata Repository*) (SUN, 2002). A utilização do repositório de metadados *MDR* adiciona à solução conformidade com as especificações *MOF* (*Meta Object Facility*) e *XMI* (*XML Metadata Interchange*), descritas no Anexo 3. É ele que fornece o suporte necessário ao armazenamento de artefatos, através do armazenamento de seus metadados e metamodelos correspondentes. Ele, também, proporciona recursos de leitura e de gravação dos artefatos em arquivos *XMI* (metadados e metamodelos). A implementação do suporte à persistência de artefatos no ambiente *DiSEN* está sendo efetuada por meio do *Distributed Artefact ReposiTory (DART)*; detalhes podem ser encontrados em (MORO, 2003).

3.2.1.5 Canal de Comunicação

Responsável pela comunicação entre as camadas da infra-estrutura (*middleware* e *middle-agent*) e a camada de aplicação. Toda comunicação entre os elementos da arquitetura é realizada por intermédio do canal de comunicação.

3.2.1.6 Gerenciador de Agentes

O gerenciador de agentes é responsável por criação, registro, localização, migração e destruição de agentes e consiste na região de agente e no servidor de ontologia. Mais detalhes sobre o gerenciador de agentes e agentes podem ser vistos em PASCUTTI (2002).

3.2.2 Desenvolvimento Cooperativo de *Software* no Contexto do *DiSEN*

O desenvolvimento de sistemas complexos demanda cooperação intensiva

entre os vários membros de um projeto com diferentes responsabilidades. O processo de desenvolvimento é freqüentemente distribuído através do tempo e do espaço e realiza-se dentro e entre grupos de trabalho especializados (ALTMANN; POMBERGER, 1999).

Ambientes de desenvolvimento que, explicitamente, suportam trabalho em grupo são um importante pré-requisito para a produção de sistemas de *software* de alta qualidade. Desse modo, uma das áreas que tem despertado o interesse de pesquisadores refere-se ao estabelecimento de estudos relacionados ao apoio efetivo do envolvimento de diversos profissionais que atuam cooperativamente no processo de desenvolvimento de *software* (REIS, 1998).

O apoio ao desenvolvimento cooperativo de *software* corresponde à disponibilização de técnicas e de ferramentas que forneçam o apoio a grupos de engenheiros de *software* nas tarefas que são realizadas cooperativamente. Assim sendo, os ambientes de desenvolvimento de *software* cooperativo surgem com o objetivo de proporcionar uma estrutura computacional que gerencie o intercâmbio de informações entre os desenvolvedores, mesmo que esses estejam em localidades geograficamente dispersas (REIS, 1998).

Vários aspectos devem ser levados em consideração para que essa cooperação seja entendida, possibilitando que os problemas gerados por ela (como, por exemplo, forma de comunicação desordenada) sejam minimizados. O processo de cooperação, ou seja, o entendimento ou as convergências de idéias entre as pessoas na realização de uma atividade de trabalho cooperativo compreende a (NUNES, 2001):

- **comunicação**: conhecimento dos canais disponíveis para troca de informações (correio eletrônico, por exemplo), para que as pessoas possam interagir mesmo estando em locais diferentes;
- **coordenação**: controle das atividades do grupo para que a cooperação ocorra de forma ordenada;
- **memória de grupo**: informações armazenadas que podem ser úteis para futuras tomadas de decisões ou simplesmente para atualização de

informações para as pessoas que estiveram ausentes do grupo por um determinado tempo;

- **percepção (*awareness*)**: refere-se a ter conhecimento das atividades do grupo. Sem *awareness*, o trabalho cooperativo coordenado é quase impossível, pois percepção é imprescindível para qualquer forma de cooperação, uma vez que perceber, reconhecer e compreender as atividades dos outros é requisito básico para a interação humana e a comunicação em geral (SOHLENKAMP, 1998).

PASCUTTI (2002, p. 90) deixa como sugestão para trabalhos futuros a definição e implementação de suporte ao trabalho cooperativo no *DiSEN*:

> "Como o *DiSEN* possui diversos *workspaces*, onde é permitida a edição cooperativa de artefatos, é necessário que esses *workspaces* sejam sincronizados e dêem suporte a *awareness*, pois é necessário levar em consideração um conjunto de questões, como, por exemplo, quais mudanças os participantes estão fazendo?, onde as mudanças estão sendo feitas?, o que os participantes podem fazer?, quais objetos os participantes estão usando?".

Sendo assim, o *DiSEN* inclui elementos que provêem apoio ao trabalho cooperativo através da criação, por exemplo, de *workspaces*. Porém o *DiSEN*, na sua versão atual, não trata, por exemplo, de elementos de percepção, constituindo-se assim em um trabalho a ser desenvolvido.

3.3 CONSIDERAÇÕES FINAIS

Neste capítulo, foram vistos os principais conceitos envolvendo a *MDSODI* e o ambiente *DiSEN*.

Neste trabalho, usaremos o *DiSEN* como ambiente e a *MDSODI* como a metodologia para a implementação da ferramenta denominada *REQUSITE*, apresentada no próximo capítulo.

4 A FERRAMENTA *REQUISITE*

A *REQUISITE* é uma ferramenta baseada em cenários que têm por objetivos auxiliar na modelagem de requisitos, prover um meio de comunicação entre os *stakeholders*, prover apoio à rastreabilidade e à documentação de requisitos no ambiente distribuído de desenvolvimento *software DiSEN*, oferecendo suporte à *MDSODI*.

A ferramenta *REQUISITE* apresenta um modelo de solução distribuída, independente de plataforma, onde vários *stakeholder* podem trabalhar de forma cooperativa, na fase de requisitos, no desenvolvimento de *software* distribuído.

Um dos pontos positivos da *REQUISITE* é o uso de técnicas baseadas em cenários. Cenários são, normalmente, facilmente compreendidos pelos diversos *stakeholders* (BREITMAN, 2000). Essa característica tem facilitado e auxiliado no trabalho de entendimento, negociação e validação dos requisitos de sistemas de *software*.

Dessa forma, a *REQUISITE*, se usada adequadamente, gera especificações que descrevem de forma não-ambígua, consistente e completa o comportamento do universo de informações do sistema (*UdI*), proporcionando aumento da qualidade e redução no tempo e no custo da atividade de definição de requisitos.

A ferramenta *REQUISITE* oferece suporte à *MDSODI* no que se refere à fase de requisitos, através da criação dos modelos de cenário e *LEL* proposta por LEITE *et al.* (1997), descrita no capítulo 2, subseção 2.4.6, e *use case* (JACOBSON, 1995) (BOOCH *et al.*, 1999), descrita no capítulo 2, subseção 2.4.5.

Neste capítulo, apresentamos o modelo de processo, os aspectos funcionais, a arquitetura, a implementação e as interfaces da ferramenta *REQUISITE*.

4.1 O MODELO DE PROCESSO

O modelo de processo de desenvolvimento da ferramenta *REQUISITE* toma, como base, o modelo de processo proposto por KOTONYA e SOMMERVILLE

(1997), descrito no capítulo 2, seção 2.3, que descreve as seguintes fases do processo de requisitos: elicitação, análise e negociação, documentação, validação e gerenciamento de requisitos. Apesar de essas atividades serem, normalmente, descritas independentemente e numa ordem particular, na prática, elas consistem de processos iterativos e inter-relacionados. A Figura 12 ilustra o modelo de processo de desenvolvimento utilizado pela *REQUISITE*.

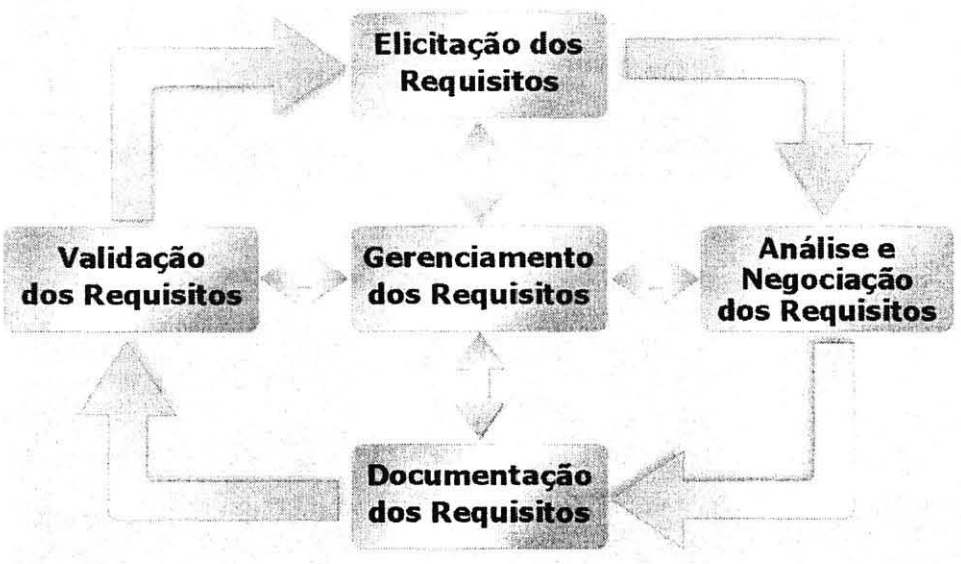


FIGURA 12 - O MODELO DE PROCESSO DA *REQUISITE*

As técnicas utilizadas pela ferramenta *REQUISITE* para o processo de engenharia de requisitos são baseadas em cenários. Cenários são utilizados não somente para modelagem, mas também para elicitação, validação e documentação de requisitos. No Anexo 1, descrevemos os processos para a construção de *LEL*, cenários e *use case*.

A elicitação dos requisitos é a fase inicial do processo, responsável por descobrir os requisitos do sistema. As técnicas de modelagem *use case* e cenários e *LEL* fornecerão um modelo das informações no *UdI*. Nessa etapa, serão identificados os atores, os *use cases*, os cenários e *LEL*.

A análise e a negociação dos requisitos é a fase do processo em que os requisitos serão analisados, a fim de se detectarem incompletudes, omissões e redundâncias.

A documentação dos requisitos é o meio através do qual será possível descrever as restrições quanto às características do *software*, quanto ao processo de desenvolvimento, a interface com outras aplicações, a descrição sobre o domínio e as informações de suporte ao conhecimento do problema. Serão produzidos, nessa etapa, os artefatos: diagrama de *use case* , cenário e *LEL*.

Na fase de validação de requisitos, será certificado se o documento de requisitos é consistente com as necessidades dos usuários. Descrever os requisitos de forma explícita é uma condição necessária não somente para validar os requisitos, mas também para resolver conflitos entre usuários.

Gerenciar requisitos significa poder escrever e acompanhar um requisito ao longo de todo o processo de desenvolvimento, enquanto esse existir. Isso garante documentos de requisitos mais confiáveis e gerenciáveis, aspectos importantes para a obtenção de produtos de *software* de qualidade. Algumas das atividades da gerência de requisitos incluem rastreamento de requisitos que podem ser executados através da exploração dos modelos, utilizando-se, por exemplo, hipertexto.

4.2 ASPECTOS FUNCIONAIS

A ferramenta apóia a modelagem de requisitos para a *MDSODI,* permitindo ao engenheiro de requisitos criar e editar diagramas de casos de uso, cenários e *LEL,* bem como rastrear requisitos. A Figura 13 apresenta o diagrama de *use case* da ferramenta *REQUISITE*.

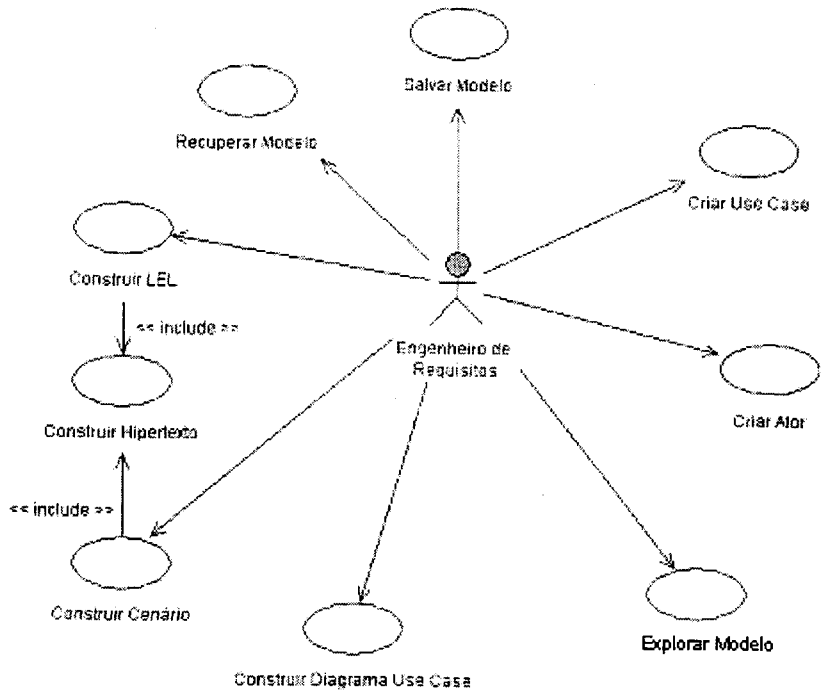


FIGURA 13 – DIAGRAMA DE *USE CASE* DA *REQUISITE*

As principais funcionalidades da ferramenta *REQUISITE* são:

a) **recuperar modelo** – esta funcionalidade permite ao engenheiro de requisitos recuperar um modelo localmente (*workspace*) ou no ambiente distribuído *DiSEN*;

b) **salvar modelo** – esta funcionalidade permite ao engenheiro de requisitos salvar um modelo localmente (*workspace*) ou no ambiente distribuído *DiSEN*;

c) **explorar modelo** – esta funcionalidade permite ao engenheiro de requisitos rastrear cenários, entradas do *LEL*, atores, *use case* e diagramas de *use case* no modelo;

d) **construir cenário** – esta funcionalidade permite ao engenheiro de requisitos criar, modificar, consultar ou remover um cenário ao modelo.

A primeira operação diz respeito à criação e à introdução de um novo cenário. De um modo geral, é o resultado fase de elicitação e que acontece com mais freqüência durante as etapas iniciais do processo. Não obstante, novos cenários podem surgir em qualquer fase de desenvolvimento, pois lembramos que um sistema de

software faz parte de um sistema maior e dessa forma, estará sempre sujeito às mudanças do ambiente onde está inserido.

A operação de modificação pode ser definida como um processo de refinamento da informação codificada em um cenário. A medida em que se desenrola o desenvolvimento de um sistema, aumenta a compreensão do ambiente onde este será inserido e de seus requisitos. Durante esse processo, a informação capturada nas fases iniciais vai sendo refinada de modo a refletir, mais precisamente, as necessidades do sistema. Em conseqüência, o conteúdo dos cenários associados sofre modificações.

A operação de retirada consiste na exclusão de um cenário do modelo;

e) **construir *LEL*** – o engenheiro de requisito, através desta funcionalidade, poderá criar, modificar, remover e consultar uma entrada no *LEL* do modelo. A primeira operação diz respeito à criação de uma nova entrada no *LEL*, isto é, a identificação de um símbolo usado na linguagem do *UdI.* As operações de modificação e remoção de uma entrada no *LEL* dizem respeito ao processo de construção do *LEL* que é continuado;

f) **criar ator** – permite ao engenheiro de requisitos adicionar, modificar, consultar e remover um ator ao modelo. Os atores previstos pela notação da *MDSODI* são: exclusivos, paralelos, distribuído e paralelo e distribuído, conforme notação descrita no capítulo 3, seção 3.1, Quadro 2;

g) **criar *use case*** – permite ao engenheiro de requisitos adicionar, modificar, consultar e remover um *use case* ao modelo. Os *use cases* previstos pela notação da *MDSODI* são: seqüencial, distribuído, paralelo e paralelo e distribuído, conforme notação descrita no capítulo 3, seção 3.1, Quadro 1;

h) **construir diagrama de *use case*** – permite ao engenheiro de requisitos criar, construir, modificar, consultar ou remover um diagrama de *use case* ao modelo.

Os relacionamentos previstos pela notação da *MDSODI* são: exclusivo e paralelo, conforme notação descrita no capítulo 3, seção 3.1, Quadro 4.

4.3 A ARQUITETURA

A arquitetura da ferramenta *REQUISITE* está apoiada em três camadas: interface, *software* base e repositório, como ilustrado pela Figura 14.

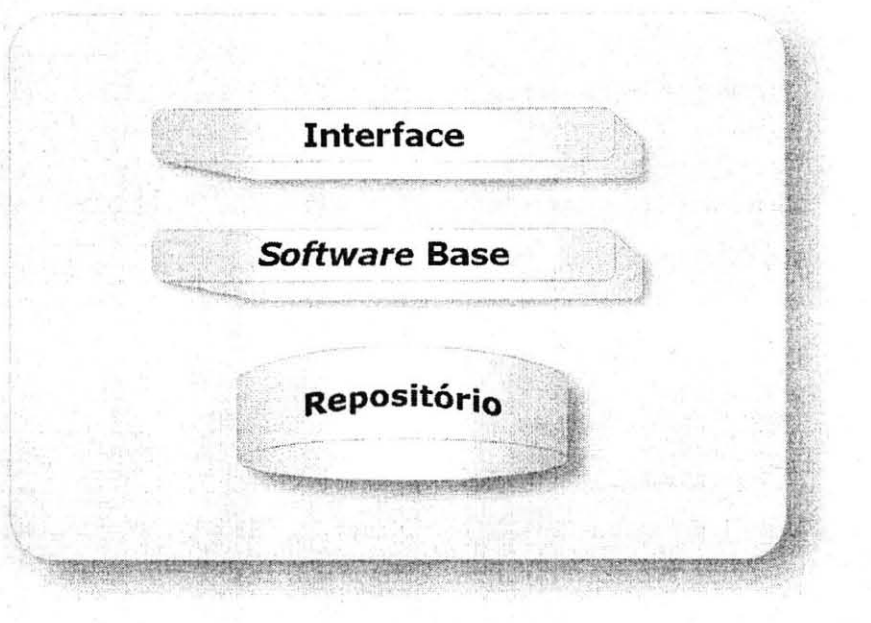


FIGURA 14 – A ARQUITETURA DA FERRAMENTA *REQUISITE*

Na camada superior, temos o *software* que faz a interface com os engenheiros de requisitos. Através dessa camada, todas as interações com o sistema serão realizadas. Essa camada é a única camada de acesso pelo engenheiro de requisitos. O acesso às duas camadas restantes são transparentes.

Na camada base, temos o *software* responsável pelo processamento das informações, ou seja, a lógica da aplicação. Por um lado, temos toda a crítica e o processamento das informações fornecidas pelo engenheiro de requisitos e, por outro, o tratamento das respostas às operações efetuadas no repositório.

Na camada inferior, temos o repositório, responsável pelo armazenamento das informações, tais como diagrama de *use case*, *use cases*, atores, cenários e *LEL*.

A arquitetura detalhada da *REQUISITE* está representada na Figura 15. Cada uma das partes constituintes da ferramenta são descritas a seguir.

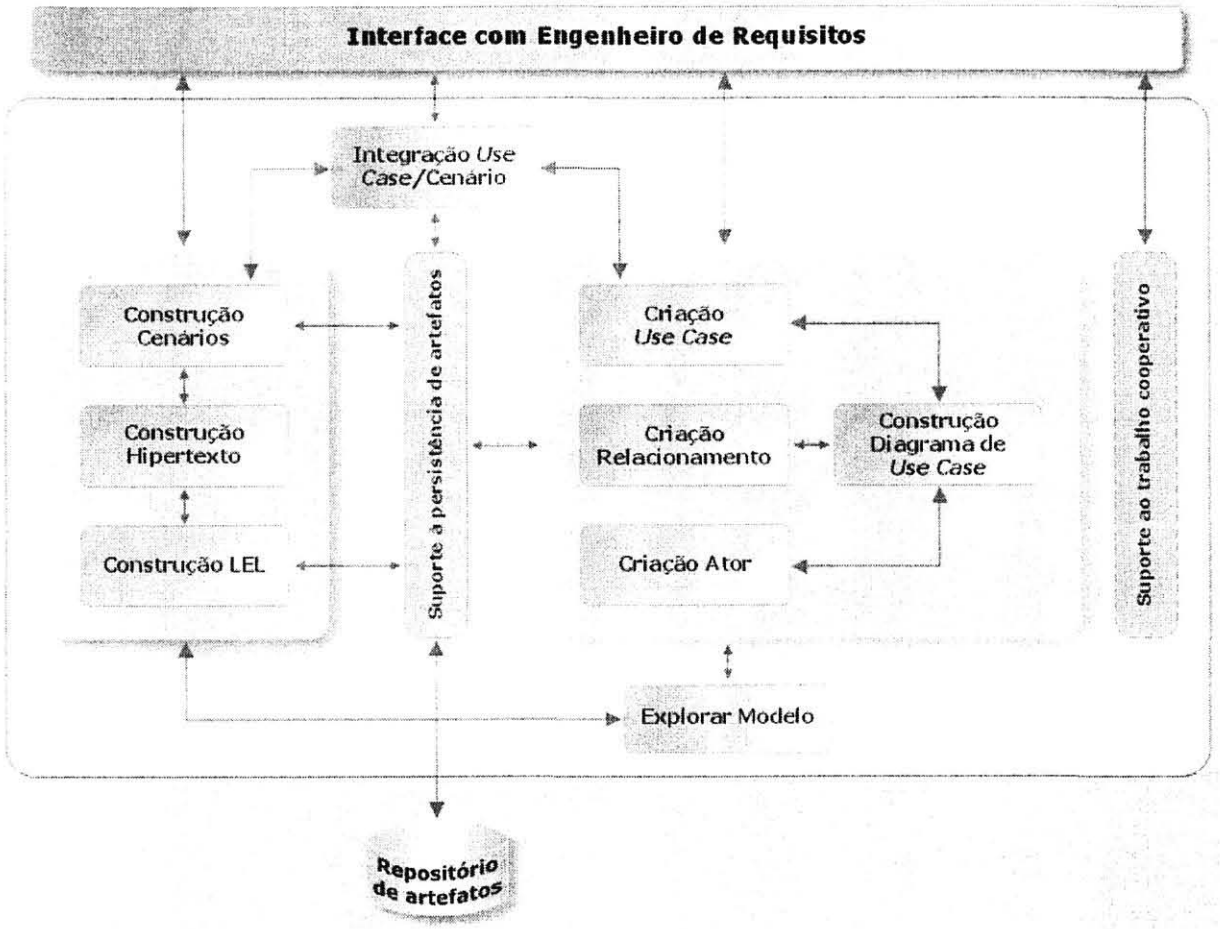


FIGURA 15 – ARQUITETURA DETALHADA DA FERRAMENTA *REQUISITE*

Interface com o Engenheiro de Requisitos – é a responsável pela comunicação entre o engenheiro de requisitos e a ferramenta, permitindo, com isso, o apoio no processo de requisitos.

Construção Cenários – permite as operações de inclusão, modificação e retirada de um cenário de um modelo.

Construção *LEL* – permite as operações de inclusão, modificação e retirada de uma entrada no *LEL*.

Construção Hipertexto – gera elos de hipertexto entre o *LEL* e os cenários. Gera também hipertexto entre um termo utilizado no *LEL* e uma entrada no *LEL* (princípio da circularidade). Esta construção permite a rastreabilidade entre os modelos.

Criação *Use Case* – permite a adição, a remoção e a modificação de um *use case*.

Criação Ator – permite a adição, a remoção e a modificação de um ator.

Criação Relacionamento – permite a adição, a remoção e a modificação de um relacionamento em um diagrama de *use case*. Os relacionamentos podem ser entre *use case* ou entre ator e *use case*.

Construção Diagrama de *Use Case* – cria e edita diagramas de *use case*. Isso significa a criação, a inclusão, a remoção, a modificação e a manipulação de atores, de *use cases* e de relacionamentos no diagrama.

Integração *Use Case* / Cenário – cria, remove e modifica uma ligação entre um caso de uso e um cenário. Através desta ligação, pode-se rastrear o cenário ligado ao caso de uso.

Explorar Modelo – explora o modelo através de uma visão em forma de árvore (*TreeExplorer*).

Suporte à Persistência de Artefatos – o suporte à persistência de artefatos permite que um engenheiro de requisitos possa armazenar, recuperar e remover artefatos de um repositório de forma transparente. O suporte à persitência torna possível disponibilizar um repositório de metadados no ambiente distribuído de desenvolvimento de *software DiSEN*, considerando requisitos que dizem respeito a acesso aos dados, disponibilidade, controle de versões, escalabilidade, transparência de localização, recuperação de falhas, desempenho, sincronização, atomicidade das operações e balanceamento de carga entre outros.

Suporte ao Trabalho Cooperativo – o suporte ao trabalho cooperativo é fornecido através de elementos de percepção que capturam e condensam as informações coletadas durante a interação entre os participantes. O suporte ao trabalho cooperativo fornece serviços de colaboração, o que torna possível a comunicação entre os membros do grupo.

4.4 EXEMPLO DE INSTÂNCIAS DA *REQUISITE* NO *DISEN*

A seguir, mostramos, na Figura 16, um exemplo ilustrativo do funcionamento da ferramenta *REQUISITE*, no ambiente *DiSEN*.

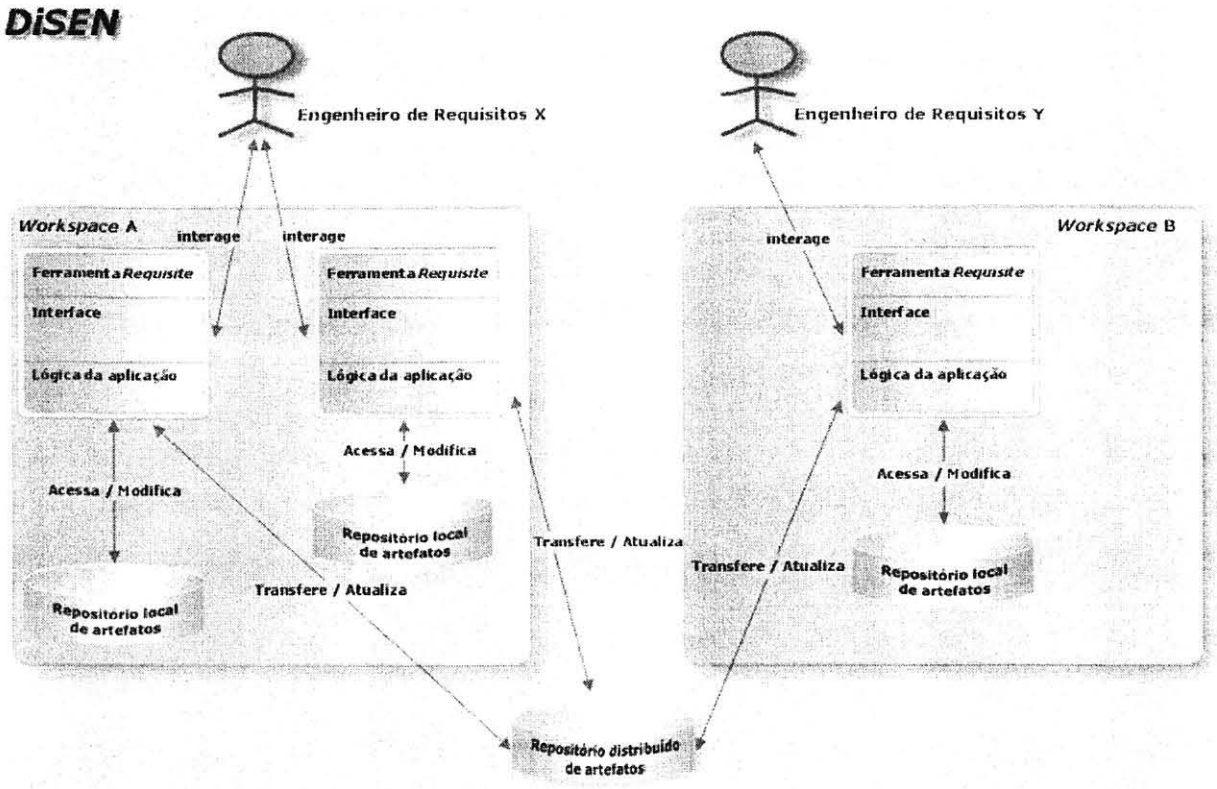


FIGURA 16 – ESQUEMA DE INSTÂNCIAS DA FERRAMENTA *REQUISITE* NO *DiSEN*

No ***Workspace*** **A**, temos o **Engenheiro de Requisitos X,** com duas instâncias da ferramenta *REQUISITE* em execução no *DiSEN*. No ***Workspace*** **B**, temos o **Engenheiro de Requisitos Y**, com uma instância da ferramenta *REQUISITE* em execução no *DiSEN*. Cada instância da ferramenta *REQUISITE* utiliza um repositório de artefato local no seu *workspace* e pode acessar, quando desejar, artefatos de um repositório de artefato distribuído no *DiSEN*.

Quando, por exemplo, o **Engenheiro de Requisitos Y** desejar alterar um artefato que está no repositório distribuído, no caso mais simples, os serviços de suporte à persistência do *DiSEN* faz uma cópia deste modelo e transfere para o repositório local. Após o término de seu trabalho, quando desejar, no caso mais simples, o **Engenheiro de Requisitos Y** ativa os serviços de suporte à persistência do *DiSEN* e este atualiza o modelo no repositório distribuído.

Desse modo, um grupo de *stakeholder*s podem agregar diferentes tipos de conhecimentos, habilidades e soluções e contribuir para que as atividades sejam finalizadas com um desempenho positivo.

Na representação dos elementos da arquitetura do *DiSEN*, ilustrado na Figura 17, destacamos que, em um *workspace*, na **Camada de Aplicação**, podem ser executadas várias aplicações. Como exemplo, podemos citar: a ferramenta *REQUISITE*, *DIMANAGER* (*Distributed Software Manager*) (PEDRAS, 2003) e *MDRExplorer* (MORO, 2003) e/ou várias instâncias da mesma aplicação como monstrado acima, Figura 16.

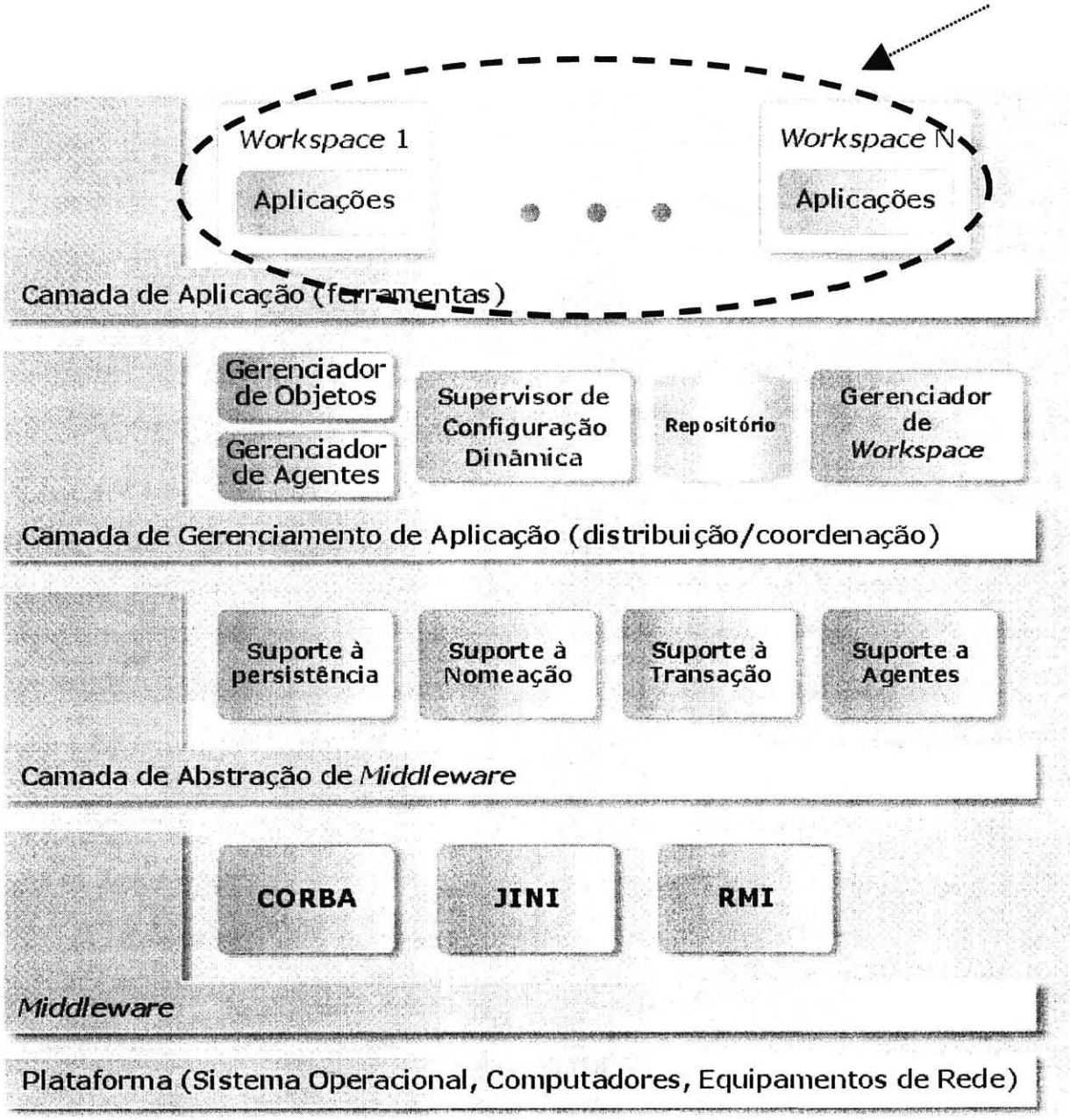


FIGURA 17 –REPRESENTAÇÃO DA ARQUITETURA *DiSEN*

4.5 IMPLEMENTAÇÃO

A ferramenta *REQUISITE* foi implementada em *Java*. A escolha dessa linguagem deve-se ao fato de ela suportar a orientação a objetos e fornecer suporte às necessidades do desenvolvimento de aplicações em um ambiente distribuído e heterogêneo, permitindo, dessa forma que a ferramenta *REQUISITE* seja executada independente de plataforma.

A ferramenta *REQUISITE* provê uma interface gráfica, para a modelagem do diagrama de *use case*. A interface gráfica da ferramenta apóia também a construção de cenários e *LEL*.

Para a persistência de artefatos, o *DiSEN* utiliza o repositório de metadados *MDR* (*Metadata Repository*) (SUN, 2002). A utilização do repositório de metadados *MDR* adiciona à solução conformidade com as especificações *MOF* (*Meta Object Facility*) e *XMI* (*XML Metadata Interchange*), descritas no Anexo 3. O *MDR* fornece o apoio necessário ao armazenamento de artefatos, através do armazenamento de seus metadados e metamodelos correspondentes. Ele, também, proporciona recursos de leitura e gravação dos artefatos em arquivos *XMI* (metadados e metamodelos). O suporte à persistência de artefatos no ambiente *DiSEN* está sendo implementado pelo *DART* (*Distributed Artefact ReposiTory*). Mais detalhes podem ser encontrados em MORO (2003) .

As classes implementadas para a ferramenta *REQUISITE* estão representadas pelo Diagrama de Classe, Figura 18. A classe ***Requisite Model*** representa o modelo gerado pela ferramenta *REQUISITE*. As classes ***Scenario*** e ***LEL*** são utilizadas para a modelagem da representação utilizada por LEITE *et al.* (1997). As classes ***Use Case***, ***Actor***, ***Association*** e ***Use Case Diagram*** são utilizadas para a modelagem da representação utilizada por JACOBSON (1995).

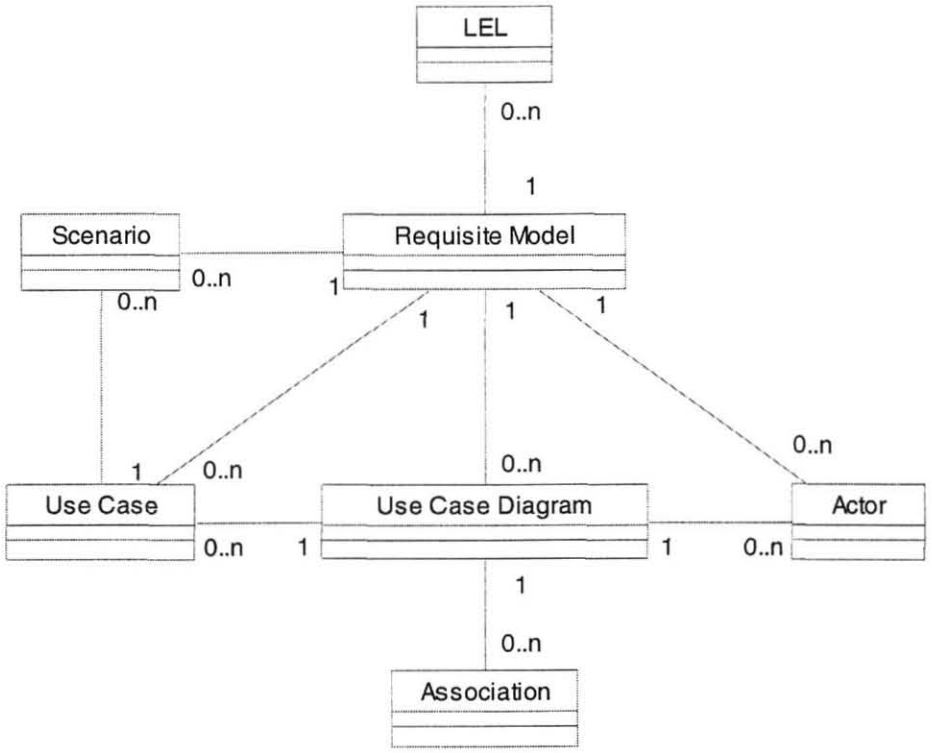


FIGURA 18 – DIAGRAMA DE CLASSE DA *REQUISITE*

4.6 INTERFACE

A seguir, apresentamos exemplos das interfaces da ferramenta *REQUISITE* . Os exemplos utilizados nesta seção são parte do estudo de caso do Sistema de Controle de Evento Científicos apresentado no capítulo 5.

A Figura 19 apresenta a janela inicial da ferramenta *REQUISITE*.

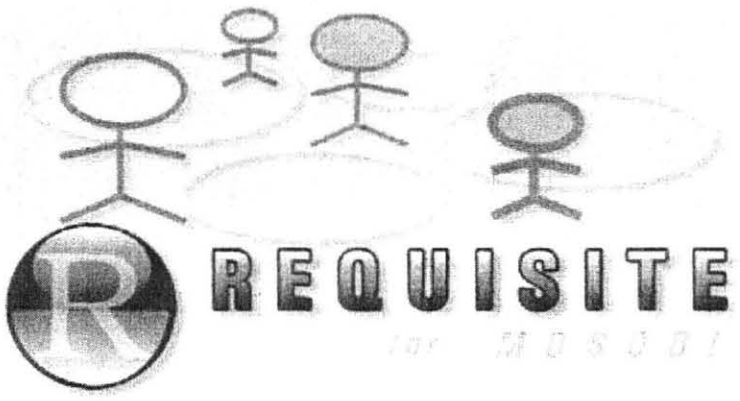


FIGURA 19 – JANELA INICIAL DA *REQUISITE*

A Figura 20 apresenta a janela utilizada pelo engenheiro de requisitos para construção do *LEL*. Os campos a serem preenchidos são: ***Name***, ***Notion*** e ***Behavioral Response***. As palavras sublinhadas são elos para outros símbolos do *LEL* (princípio da circularidade).

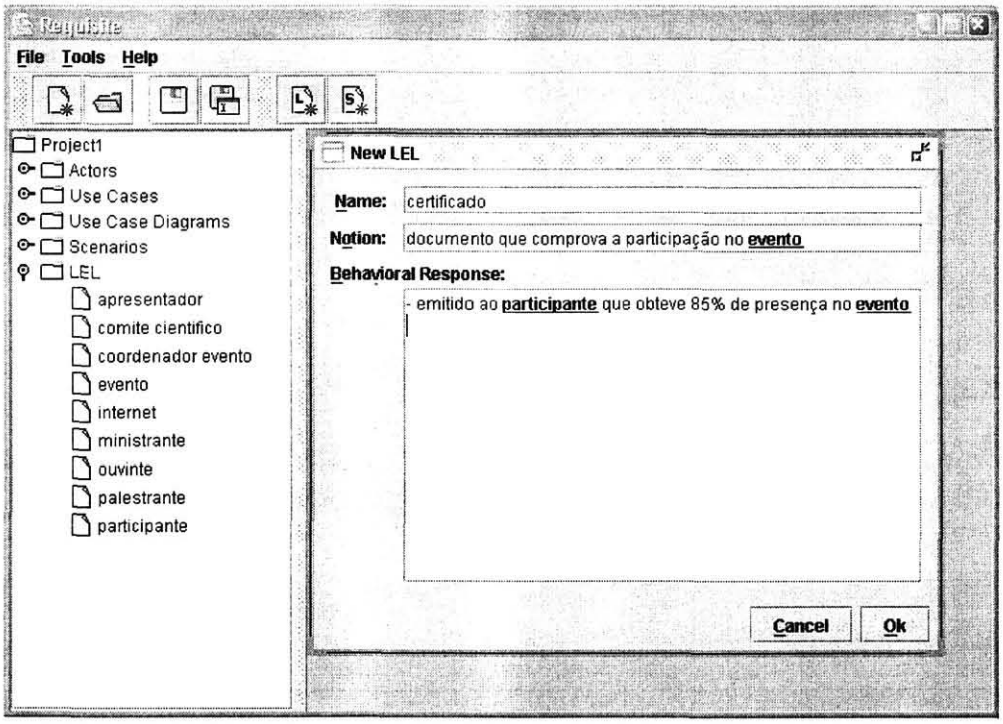


FIGURA 20 – JANELA DE CONSTRUÇÃO DO *LEL*

A Figura 21 apresenta a janela utilizada pelo engenheiro de requisitos para construção de cenários. Os campos a serem preenchido são: ***Title***, ***Goal***, ***Context***, ***Resources***, ***Actors*** e ***Episodes***. As palavras sublinhadas são elos para símbolos do *LEL*.

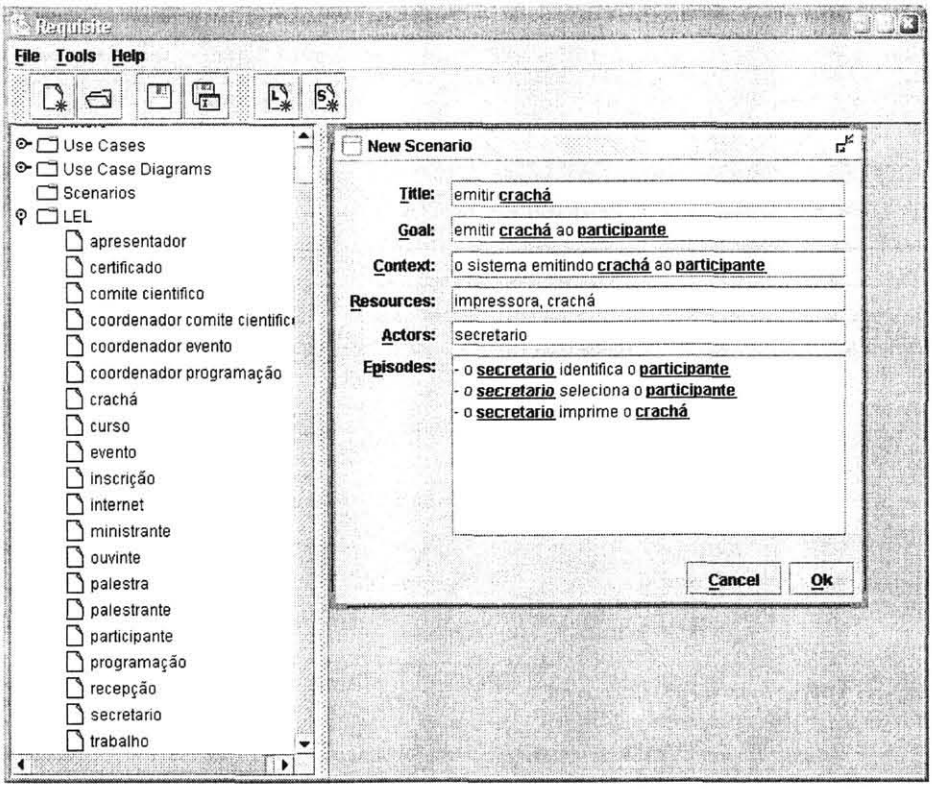


FIGURA 21 – JANELA DE CONSTRUÇÃO DE CENÁRIOS

A Figura 22 apresenta a janela utilizada pelo engenheiro de requisitos para construção de Diagramas de *Use Case*. Podemos observar, através da figura, a notação da *MDSODI* para atores e *use case*.

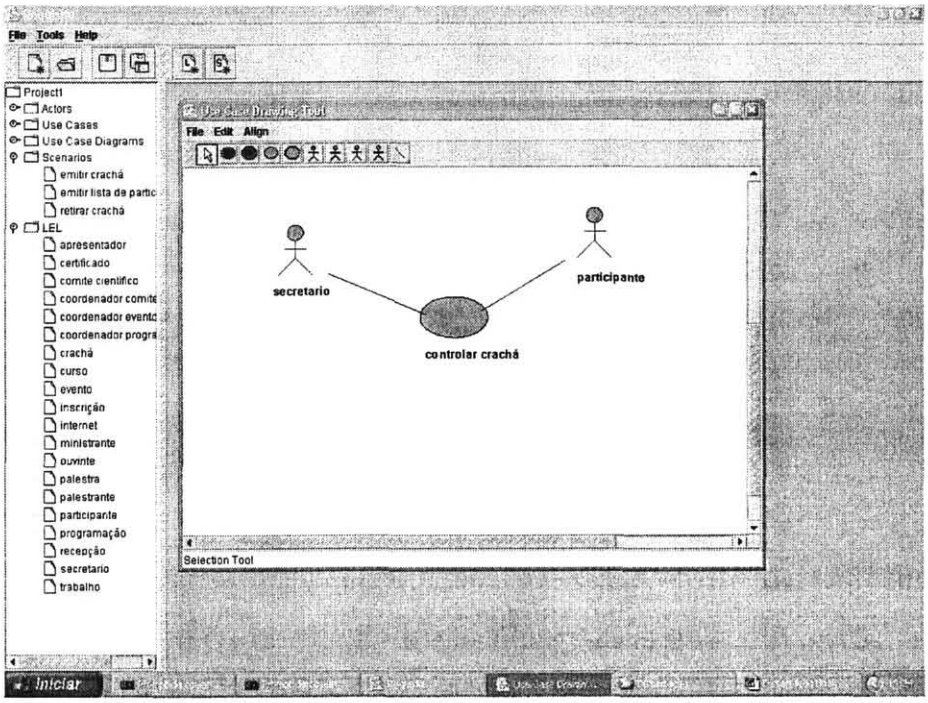


FIGURA 22 – JANELA DE CONSTRUÇÃO DO DIAGRAMA DE *USE CASE*

4.7 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentadas as funionalidades, a arquitetura, a implementação e a interface da ferramenta *REQUISITE*.

A ferramenta *REQUISITE*, se usada adequadamente, auxilia na modelagem de requisitos, provê um meio de comunicação entre os *stakeholders*, provê apoio à rastreabilidade e à documentação de requisitos.

No próximo capítulo, apresentamos um estudo de caso para validar a *REQUISITE*.

5 VALIDAÇÃO DA FERRAMENTA *REQUISITE*

Para a validação da ferramenta *REQUISITE*, escolhemos para o estudo de caso um sistema de Controle de Eventos Científicos que tem por objetivo manter e controlar informações de eventos científicos. Através deste estudo de caso, destacamos as funcionalidades da ferramenta *REQUISITE* bem como o processo de modelagem baseado em cenários, descritos no Anexo 1.

Neste capítulo, apresentamos o sistema exemplo e a construção do sistema exemplo com o apoio da ferramenta *REQUISITE.*

5.1 O SISTEMA EXEMPLO: CONTROLE DE EVENTOS CIENTÍFICOS

Uma pessoa que esteja interessada em participar de um evento poderá fazer sua inscrição via *Internet* ou na recepção do evento.

O participante poderá fazer sua inscrição em uma das seguintes categorias: ouvinte, palestrante, ministrante ou apresentador de trabalho. O ouvinte é aquele que se inscreve apenas para assistir ao evento. O apresentador é aquele que se inscreve para apresentar um ou mais trabalhos. O palestrante é convidado pela comissão organizadora para realizar uma ou mais palestras durante o evento. O ministrante é convidado pela comissão organizadora para ministrar um ou mais cursos durante o evento.

A comissão organizadora elaborará a programação do evento com base nos trabalhos, cursos e palestras, previamente selecionados pelo comitê científico.

Crachás de identificação devem ser entregues, na recepção do evento, a todos os participantes. Só será permitido o acesso ao evento aos portadores de crachás.

Para comprovar a participação no evento, os participantes receberão certificados de participação. Os certificados só deverão ser emitidos para os participantes com, no mínimo, 85% de presença.

5.2 A CONSTRUÇÃO DO MODELO COM O APOIO DA *REQUISITE*

A seguir, descrevemos a construção do modelo do estudo de caso com o apoio da ferramenta *REQUSITE*. Os processos de construção das técnicas baseadas em cenários estão descritos no Anexo 1. Apesar de as atividades do processo de construção dos modelos estarem descritas independentemente e numa ordem particular, na prática, elas consistem de processos iterativos e inter-relacionados. Esse processo foi realizado adotando-se o princípio de desenvolvimento de *software* iterativo e incremental definido pela *MDSODI.*

O primeiro passo foi a identificação das entrada no *LEL*, ou seja, a elicitação da linguagem usada no *UdI*. Identificamos os termos usados no *UdI* e contruímos o *LEL*. Para essa construção, foi utilizado a funcionalidade ***New LEL***.

As entradas do *LEL* encontradas para este exemplo foram: participante, ouvinte, coordenador, palestrante, ministrante, apresentador, evento, Internet, crachá, comitê científico, comitê organizador, certificado, programação, trabalho, curso, palestra, inscrição, recepção, secretário.

A Figura 23 apresenta a janela da ferramenta *REQUISITE* para a construção do *LEL:* **crachá**. Os campos preenchido são: ***Name***, ***Notion*** e ***Behavioral Response***. As palavras sublinhadas são elos para outros símbolos do *LEL* (princípio da circularidade).

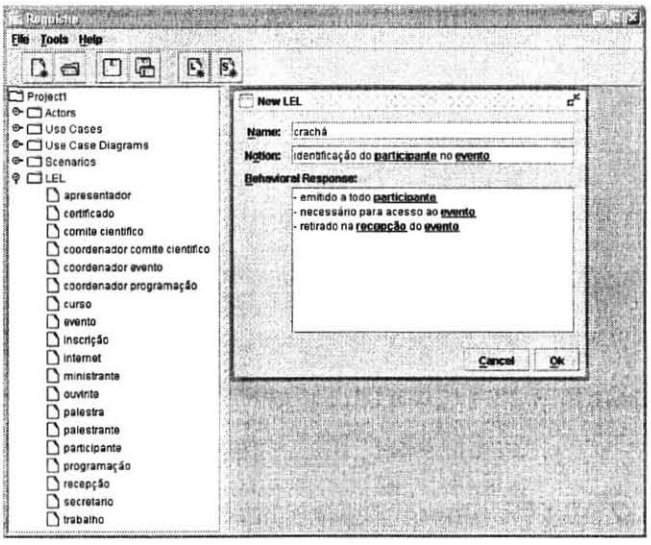


FIGURA 23- CONSTRUÇÃO DO *LEL:* CRACHÁ

No segundo passo, foram descobertos os atores e prosseguiu-se com a descoberta dos *use cases* associados a esses atores. Para cada ator, foram encontrados os *use cases* relacionados ao mesmo.

Para esta etapa, foram utilizadas as funcionalidade ***New Actor*** e ***New Use Case*** respectivamente, considerando a *MDSODI*, ou seja, aspectos de sistema distribuídos paralelismo/concorrência e distribuição, desde as fases iniciais do sistema.

Como resultado desse processo, foram encontrados os atores: participante, secretário, coordenador do comitê científico, coordenador de programação, coordenador do evento. Os *use cases* encontrados foram: controlar crachá, controlar participante, controlar centificado, gerenciar evento, gerenciar programação e cadastrar atividade.

O terceiro passo consistiu em definir os caminhos básicos (cenários primários) e, posteriormente, os caminhos alternativos (cenários secundários) para cada um dos *use case*. Para a construção dos cenários, foi utilizada a funcionalidade ***New Scenario***.

Para exemplificar a construção de cenário, escolhemos o *use case* **Controlar Crachá**. Para este *use case,* formam identificados os cenários: **emitir crachá**, **retirar crachá** e **emitir lista de participante**. As Figuras 24, 25 e 26 representam, respectivamente, as janelas de construção desses cenários. Os campos preenchido são: ***title***, ***goal***, ***context***, ***resources***, ***actors***, ***episodes***. As palavras sublinhadas são elos para símbolos do *LEL*.

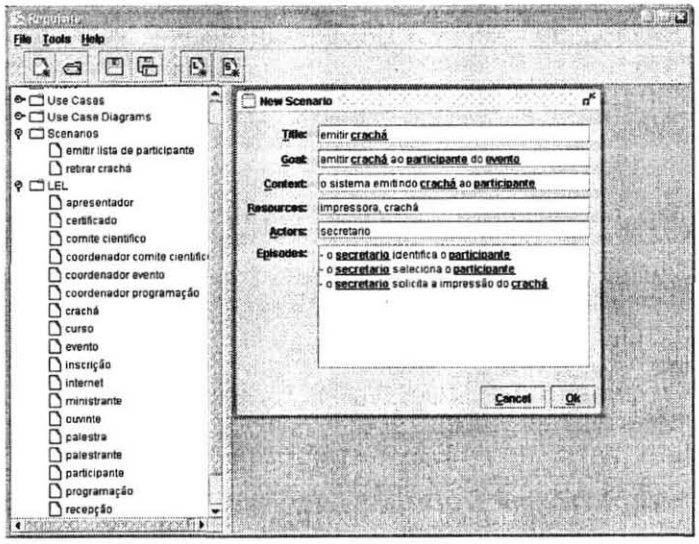


FIGURA 24 - CONSTRUÇÃO DO CENÁRIO: EMITIR CRACHÁ

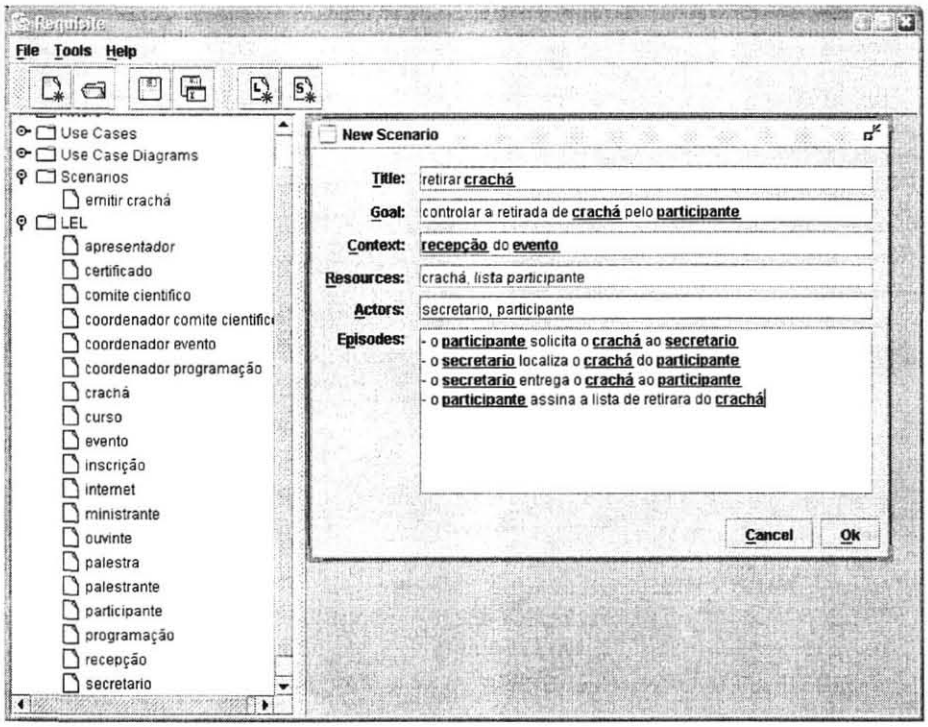


FIGURA 25 - CONSTRUÇÃO DO CENÁRIO: RETIRAR CRACHÁ

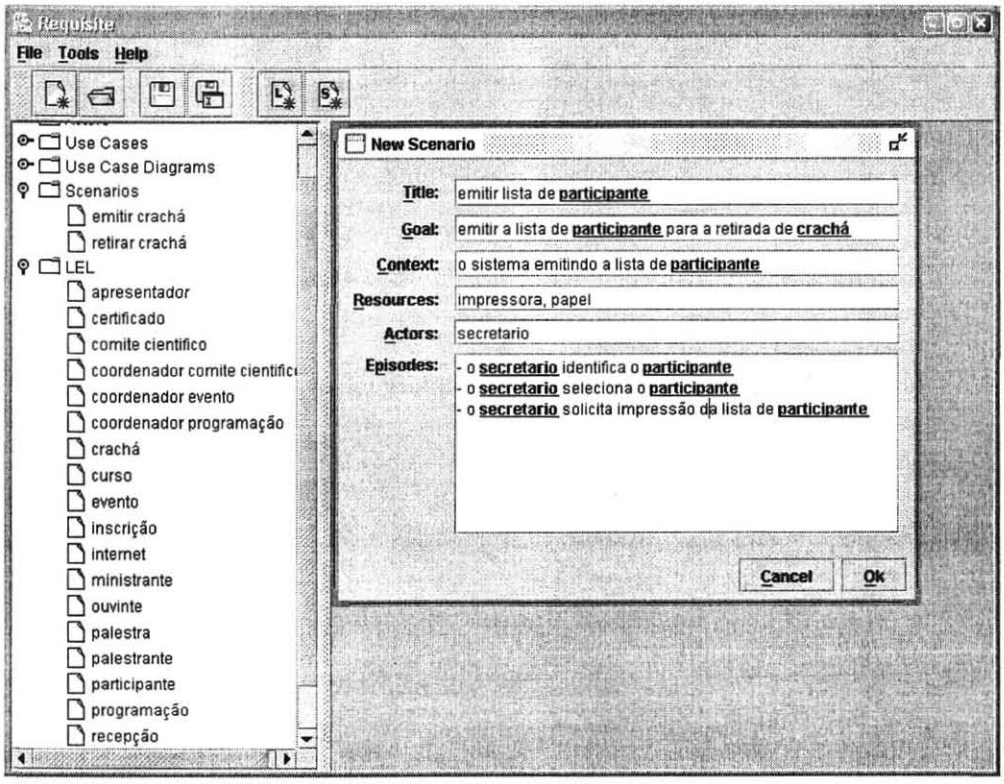


FIGURA 26 - CONSTRUÇÃO DO CENÁRIO: LISTAR PARTICIPANTE

Após definidos os *use cases* e os atores do sistema, desenvolvemos o Diagrama de *Use Case,* através da funcionalidade ***New Use Case Diagram,*** utilizando as notações da *MDSODI* apresentadas na Figura 27.

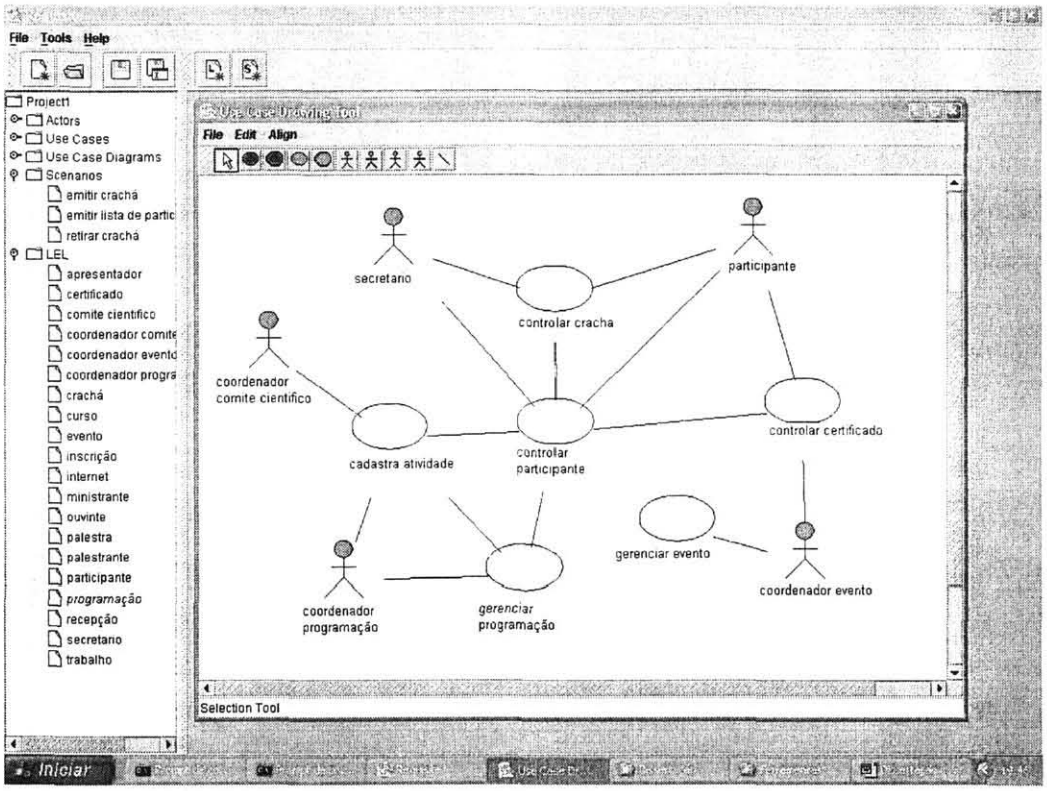


FIGURA 27 - CONSTRUÇÃO DO DIAGRAMA DE *USE CASE*

A qualquer momento do processo descrito acima, podemos armazenar o modelo no ambiente *DiSEN*, bastando, para isso, selecionar a opção no menu ***Save to DiSEN***. Para armazenar o modelo localmente no *workspace*, basta selecionar a opção no menu ***Save***. Da mesma forma, para recuperar o modelo do ambiente *DiSEN*, basta selecionar a opção no menu ***Open from DiSEN*** e, para recuperar o modelo localmente no *workspace*, basta selecionar a opção no menu ***Open***, Figura 28.

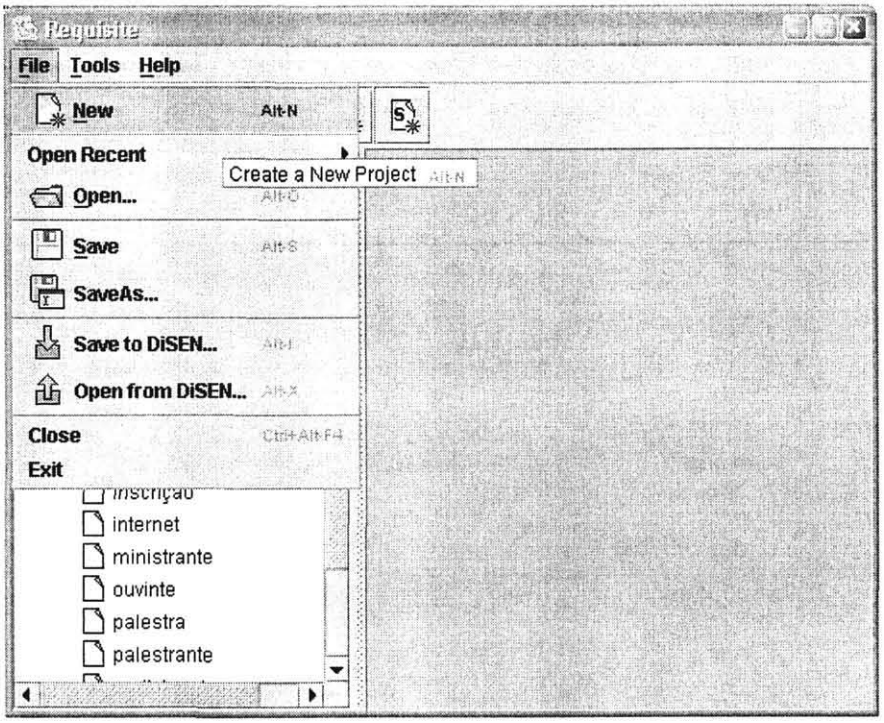


FIGURA 28 – SALVAR MODELO

5.3 CONSIDERAÇÕES FINAIS

A *REQUISITE* oferece um ambiente gráfico para a construção dos modelos. Todos os modelos podem ser rastreados, por exemplo, utilizando o hipertexto, durante e depois de sua construção. A ferramenta oferece uma documentação para as outras fases do desenvolvimento de *software*. Além disso, a *REQUISITE* respeita a organização do modelo de processo, facilitando a verificação dos requisitos do sistema.

Neste capítulo, apresentamos um estudo de caso utilizando a ferramenta *REQUISITE*, o sistema exemplo escolhido foi o de Controle de Eventos Científicos.

Finalmente, no próximo capítulo, apresentamos as conclusões, os trabalhos em andamento e os trabalhos futuros.

6 CONCLUSÃO

Neste capítulo, são apresentadas as conclusões, as contribuições, os trabalhos em andamento e, finalmente, são propostos os trabalhos futuros de pesquisas relacionados a este trabalho.

6.1 CONCLUSÕES

A crescente complexidade das aplicações, a contínua evolução tecnológica e o uso cada vez mais disseminado de redes de computadores têm incentivado os estudos referentes ao desenvolvimento de sistemas distribuídos.

Sistemas distribuídos são bastante complexos, o que, conseqüentemente, reflete na complexidade de desenvolvimento do *software*. Para que o desenvolvimento de *software* distribuído seja uma tarefa produtiva, gerando também produtos de qualidade, é necessário que o ambiente de apoio ao desenvolvedor seja estruturado de modo a prover recursos que o auxiliem na realização do processo. Técnicas, ferramentas e metodologias que ofereçam o suporte necessário ao desenvolvimento de *software* distribuído é de grande valia.

Visando suprir a necessidade de ferramentas e de ambientes de desenvolvimento distribuído de *software* foram desenvolvidos a metodologia *MDSODI* (GRAVENA, 2000) e o ambiente *DiSEN* (PASCUTTI, 2002).

Buscando oferecer suporte automatizado à fase de requisito da *MDSODI*, no contexto do *DiSEN*, definimos e implementamos a ferramenta *REQUISITE*.

A fase de requisitos tem sido reconhecida como uma fase crítica do processo da engenharia de *software*. Tal reconhecimento decorre da descoberta de que a maior parte dos problemas, geralmente os mais dispendiosos e de maior impacto negativo no desenvolvimento de *software*, são originados nas etapas iniciais do desenvolvimento, ou seja, na fase de requisitos.

A ferramenta *REQUISITE* auxilia na modelagem de requisitos, provê um meio de comunicação entre os *stakeholders*, provê apoio à rastreabilidade e à documentação

de requisitos no ambiente distribuído de desenvolvimento *software DiSEN*, dando suporte à *MDSODI*.

Dessa forma, a *REQUISITE*, se usada adequadamente, gera especificações que descrevem de forma não ambígua, consistente e completa o comportamento do universo de informações do sistema (*UdI*), proporcionando aumento da qualidade e redução no tempo e no custo da atividade de definição de requisitos.

6.2 CONTRIBUIÇÕES

A principal contribuição deste trabalho reside no desenvolvimento da ferramenta *REQUISITE*, que tem por objetivo dar suporte à fase de requisitos da *MDSODI*, no contexto do ambiente distribuído de desenvolvimento *software DiSEN*.

A ferramenta REQUISITE contribui para:

- auxiliar na modelagem de requisitos através de técnicas baseadas em cenários;
- oferecer um ambiente gráfico para a construção dos modelos;
- prover um meio de comunicação entre os *stakeholders*;
- rastrear requisitos nos modelos durante e depois de sua construção através, por exemplo, de hipertexto;
- oferecer uma documentação para as outras fases do sistema modelado;
- respeitar a organização do modelo de processo, facilitando a verificação dos requisitos do sistema.

6.3 TRABALHOS EM ANDAMENTO

- Definição e elaboração de uma ferramenta que dará apoio ao gerenciamento de desenvolvimento de *software* em sistemas distribuídos, tendo como objetivos: acompanhar o desenvolvimento do projeto através da comparação entre o planejado e o executado; verificar a situação de cada atividade, assim como das equipes envolvidas; controlar as versões

de cada atividade e os aspectos inerentes ao desenvolvimento de sistemas distribuídos. Essa ferramenta, denominada *DIMANAGER* (*Distributed Software Manager*), oferecerá suporte à metodologia *MDSODI* e também será integrada ao *DiSEN* (PEDRAS, 2003).

- Definição e implementação do *DART* (*Distributed Artefact ReposiTory*) que é um repositório distribuído de artefatos, baseado em um repositório de metadados, para o ambiente *DiSEN,* com suporte aos padrões *MOF* e *XMI* (MORO, 2003).
- Definição de um mecanismo que ofereça suporte ao gerenciamento de projetos, dentro de um ambiente compartilhado, especificamente em relação ao planejamento de projetos; atendendo às necessidades relativas ao gerenciamento de recursos humanos, quanto às características de disponibilidade, habilidades e conhecimento dos participantes de um projeto. O objetivo é oferecer ao gerente de projetos informações concisas e claras sobre as características dos indivíduos que possam vir a fazer parte de um projeto sob sua supervisão (LIMA, 2003).
- Estudos de técnicas que ofereçam suporte necessário ao tratamento de cooperação e de coordenação em desenvolvimento de sistemas multi-agentes (ONO, 2003).

6.4 TRABALHOS FUTUROS

- Integração com a ferramenta *DIMANAGER* (PEDRAS, 2003). Para possibilitar o gerenciamento do projeto e de requisitos.
- Implementação de suporte ao trabalho cooperativo. Pois, para possibilitar a cooperação, são necessárias informações sobre o que está acontecendo. Essas informações são fornecidas através de elementos de percepção que capturam e condensam as informações coletadas durante a interação entre os participantes. O suporte ao trabalho cooperativo fornecerá serviços de colaboração, o que tornará possível a comunicação entre os

membros do grupo. Trabalhando cooperativamente, pelo menos potencialmente, podem-se produzir melhores resultados do que se os membros do grupo atuassem individualmente. Em um grupo, podem ocorrer a complementação de capacidades, de conhecimento e de esforços individuais e a interação entre pessoas com entendimento, pontos de vista e habilidades complementares.

- Integração com as ferramentas de outras fases do processo de desenvolvimento de *software* definidos pela *MDSODI* (análise, projeto, implementação e teste).
- Estudo de técnicas de inteligência artificial que possibilitem a construção de uma base de conhecimento para apoiar a fase de requisitos da *MDSODI*.

REFERÊNCIAS

ALTMANN, J.; POMBERGER, G. **Cooperative *Software* Development: Concepts, Model and Tools**. In: TOOLS-30 CONFERENCE, 1999, Santa Barbara. Santa Barbara: IEEE Society Press. Proceedings ..., 1999.

AMBRIOLA, V.; GERVASI, V. **Processing Natural Language Requirements**. In: 12 th International Conference on Automated *Software* Engineering. Lake Tahoe, EUA, Proceedings ..., Nov. 1997.

ANTONIOU, G. The Role of Non-monotonic Representations in Requirements Engineering. **International Journal of *Software* Engineering and Knowledge Engineering**, vol 8. 1998.

BOOCH, G., JACOBSON, I., RUMBAUGH, J. **The Unified Modeling Language User Guide**. Addison-Wesley, 1999.

BORLAND *SOFTWARE* CORPORATION. ***CaliberRM***. Disponível em http://www.tbi.com, acesso em: 16 jun. 2003.

BREITMAN, K.K. **Evolução de Cenários.** Tese (Doutorado), Departamento de Informática PUC-Rio, maio 2000. 153p.

BROOKS, F. No Silver Bullet: Essence and Accidents of *Software* Engineering. **IEEE Computer**, Vol. 20 No. 4, April 1987, p. 10-19.

CASTRO, J. GAUTREAU, J. and TORANZO, M. **Tool Suport for Requirements Formalization**. In: ACM SIGSOFT Viewpoints 96: International Worskhop on Multiple Perspective *Software* Development - ACM VP 96. San Francisco, USA, Proceedings ..., 1996, p. 202-206.

CHUNG, L. NIXON, B. YU, E. and MYLOPOULOS, J. **Non-Functional Requirements.** in *Software* Engineering. Kluwer Academic Publisher, 2000.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed Systems: Concepts and Design**. 3rd Edition. Englad: Pearson Education, 2001.

CYSNEIROS, L.M. and LEITE, J.C.S.P. **Integrating Non-Funcional Requirements into Data Model.** In: 4th Intenational Symposium on Requirements Engineering – Ireland, Proceedings ..., June 1999.

CYSNEIROS, L.M. **Requisitos Não Funcionais: Da elicitação ao Modelo Conceitual**. Ph.D. Thesis, Departamento de Informática PUC-Rio, fev 2001. 224p.

DAMIAN,A.; HONG, D. e LI, H. **Joint Application Development and Participatory Design.** Disponível em http://www.cpsc.ucalgary.ca, acesso em: 02 dez. 2002.

DARDENE, A., Lamsweerde, V., Fikas, S. Goal-Directed Requirements Acquisition. **Science of Computer Programming**, 20, p. 3-50, 1993.

DAVIS, A. Operational Prototyping: A New Development Approach. **IEEE *Software***, 1992.

DAVIS, A. M. ***Software* Requirements: Objects, Functions, and States**. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1993.

DUBOIS, E. and HEYMANS, P. Scenario Based Techniques for Supporting the Elabotation and the Validation of Formal Requirements. **Requirements Engineering Journal** – Vol. 3 n. 3&4 , 1998. p. 202-218.

EDS – ELETRONIC DATA SYSTEM CORPORATION. ***Slate Require***. Disponível em http://www.eds.com , acesso em:16 jun. 2003.

FILLIPIDOU, D. Edesigning with Scenarios: a critical view on current research and practice. **Requirements Engineering Journal** – edited by Springer Verlag – Vol. 3 n. 1 – p. 1-22, 1998.

GALEGHER, J. and KRAUT, R. **Intelectual Teamwork – Social Fundations of Cooperative Work**. Lawrence Erlbaum Associates Publishesrs, 1990.

GOTEL, O. and FINKELNSTEIN, A. **An Analysis of the Requireme nts Traceability Problem.** In: International Conference on Requirements Engineering, Colorado Springs, EUA, Proceedings ..., Apr. 1994.

GRAVENA, J. P. **Aspectos Importantes de uma Metodologia para Desenvolvimento de *Software* com Objetos Distribuídos**. Trabalho de Graduação, Universidade Estadual de Maringá, Departamento de Informática, Maringá - PR, 2000.

HADAD, G. et. al. **Construcción de Escenarios a partir del Léxico Extendido Del Lenguage**. In: JAIIO'97, Buenos Aires, Proceedings ..., 1997, pp. 65-77.

HADAD, G. et. al. **Enfoque Middle-out en la Construcción e Integración de Scenario**. In: Workshop de Engenharia de Requisitos, Buenos Aires, Argentina, 1999, Anais, p. 79-94.

HAUMER, P., POHL, K., WEIDENHAU'PT, K. Requirements Elicitation and Validation with Real World Scenes. **IEEE Transactions on *Software* Engineering**, Vol 24, No 12, Special Issue on Scenario Management, December, 1998.

HSIA, P. et al. Formal Approach to Scenario Analysis. **IEEE *Software***, vol. 11 n.2, 1994. p.33-41.

HUZITA, E.H.M. **MOOPP - Uma Metodologia para Auxiliar o Desenvolvimento de Aplicações para Processamento Paralelo.**. Tese (Doutorado) – Escola Politécnica, USP, São Paulo. 1995. 213p.

HUZITA, E.H.M. **Uma Metodologia de Desenvolvimento Baseado em Objetos Distribuídos Inteligentes**. Projeto de Pesquisa em andamento, Universidade Estadual de Maringá. Departamento de Informática. Maringá, 1999.

IEEE Std. 830. **IEEE Guide to *Software* Requirements Specification**. The Institute of Electrical and Electronics Engineers. New York, EUA.1984.

IEEE, **IEEE *Software* Engineering Standards Collection**. Computer Society Press, 1997.

IGATECH SYSTEMS. ***RDT***. Disponível em http://www.igatech.com, acesso em: 16 jun. 2003.

INTEGRATE CHIPWARE. ***icCONCEPT RTM***. Disponível em http://www.chipware.com, acesso em: 16 jun. 2003.

JACKSON, M. ***Software* Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices.** 1ed. Addison-Wesley. Massachusetts, USA. 1995.

JACOBSON, I. **Object Oriented *Software* Engineering: A Use-*CASE* Driven Approach**. Addison-Wesley ,1995.

JACOBSON, I.; BOOCH, G.; RUMBAUCH, J. **The Unified *Software* Development Process.** Addison-Wesley Publishing Company, 1999.

KOTONYA, G.; SOMMERVILLE, I. **Requirements Engineering – Processes and Techniques**. John Willy & Sons, 1997.

KUUTTI, K. **Work Processes: Scenarios as a Preliminary Vocabulary**. in Scenario Based Design: Envisioning Work and Tecnology in System Development- John Wiley and Sons, 1995. p. 19-36.

LAMSWEERDE, A. DARIMONT, R. e LETIER, E. **Managing Conflicts in Goal-Driven Requirements Engineering.** IEEE Transactions on *Software* Engineering, Special Issue on Managing Inconsistency in *Software* Development. Nov. 1998.

LEITE, J.C.S.P. **A Survey on Requirements Analysis, Technical Report**. Department of Information and Computer Science, California, 1987.

LEITE, J.C.S.P. and FRANCO, A.P.M. **A Strategy for Conceptual Model Acquisition.** In: First IEEE International Symposium on Requirements Engineering, SanDiego, Ca, IEEE Computer Society Press, Proceedings..., 1993, p. 243-246.

LEITE, J.C.S.P.; ROSSI, G.; MAIORANA, V.; BALAGUER, F.; KAPLAN, G.; HADAD, G.; OLIVEROS, A. **Enhancing a Requirements Baseline with Scenarios.** In: Third IEEE International Symposium on Requirements Engineering - IEEE Computer Society Press. Los Alamitos, Ca, USA, Proceedings ..., 1997, p. 44 -53.

LEITE, J.C.S.P.; HADAD, G.D.S; DOORN, J.H.; KAPLAN, G.N. **A Scenario Construction Process.** Requirements Engineering Journal. 5:38-61, 2000.

LIMA F. de. **Aspectos de Gerenciamento de Recusos Humanos no Desenvolvimento Distribuído de *Software*.** Maringá, DIN 2003. Exame de Qualificação Mestrado.

LOUCOPOULOS, P. and KARAKOSTAS, V. **System Requirements Engineering**. London McGraw-Hill, 1995.

LUTZ, R.R. **Analysing *Software* Errors in Safety-Critical Embedded Systems. Proceedings of the ACM SIGSOFT.** Symposium on the Foundations od *Software* Engineering, New York, NY, December 1993, p. 126-133.

MACAULY, L.A. **Requirements Engineering**. London, 1996.

MAIDEN, N. **CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements**. Automated *Software* Engineering, 1998.

MAMANI M.; LEITE, J.C.S.P. **Elicit@99 um Protótipo de Ferramenta para a Elicitacao de Requisitos**. In: WER99 - Workshop en Requerimientos / Workshop de Engenharia de Requisitos - II. SADIO. Buenos Aires, Proceedings ..., 1999, p. 45-55.

MORO, C.F. **Proposta de um Repositório de Metadados para Ambiente de Desenvolvimento de *Software* Distribuído**. Exame de Qualificação – Mestrado em Informática. Maringá: DIN-UEM/UFPR, 2002.

MORO, C.F. **Suporte à Persistência de Artefatos para o Ambiente Distribuído de Desenvolvimento de *Software DiSEN*.** Dissertação de Mestrado em Informática. Maringá: DIN-UEM/UFPR, 2003. 100 p.

NETO, J.S.M. **Integrando Requisitos Não Funcionais ao Modelo de Objetos**. Dissertação de Mestrado, Departamento de Informática PUC-Rio, Mar 2000.

NUNES, I. D. **Componentes de Percepção para o Ambiente PROSOFT Cooperativo**. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2001. 136 p.

NUSEIBEH, B. and EASTERBROOK, S. **Requirements Engineering: A Roadmap**. In: 22 nd International Conference on *Software* Engineering. Limerick, Ireland. Proceedings..., Jun. 2000.

ONO, W.M. **Estudo do Aspecto de Cooperação no Desenvolvimento de Sistemas Multi-Agentes.** Trabalho de Iniciação Científica. Departamento de Informática. Universidade Estadual de Maringá, 2003.

PASCUTTI, M. **Uma Proposta de Arquitetura de um Ambiente de Desenvolvimento de *Software* Distribuído Baseado e Agentes**. Dissertação de Mestrado em Ciência da Computação. Instituto de Informática. Universidade Federal do Rio Grande do Sul- RS, 2002.

PEDRAS, M.E.V. **Uma Ferramenta de Apoio ao Gerenciamento de Desenvolvimento de *Software* Distribuído.** Dissertação de Mestrado em Informática. Maringá: DIN-UEM/UFPR, 2003. 113 p.

POTTS, C., TAKAHASHI, K., ANTÓN, A. Inquiry-Based Requirements Analysis. **IEEE *Software***, March 1994. p. 21-32.

PRESSMAN, R. **Software Engineering**: **A Practioner's Approach**. 5th Edition, McGraw-Hill, 2001.

RALYTÉ, J. **Reusing Scenario Based Approaches in Requirements Enginnering Methods: Crews Method Base.** In: International Workshop on the Requirements Engineering Process, Florence, Italy, Proceedings ..., September 1999, (CREWS Report Series 99-12).

RATIONAL CORPORATION. ***RequisitePro***. Disponível em http://www.rational.com, acesso em: 16 jun. 2003.

REIS, R. Q. **Uma Proposta de Suporte ao Desenvolvimento Cooperativo de *Software* no Ambiente PROSOFT.** Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 1998. 177p.

ROLLAND, C., ACHOUR, B., CAUVET, C., RALYTÉ, J., SUTCLIFFE, A., MAIDEN, N., JARKE, M., HAUMER, P., POHL, K., DUBOIS, E., HEYMANS, P. A Proposal for a Scenarios Classification Framework. **Journal of Requeriments Engineering** – vol. 3 – Springer Verlag, 1998a. p. 23-47.

ROLLAND, C., SOUVEYET, C., ACHOUR, C. B. Guiding Goal Modeling Using Scenarios. **IEEE Transactions on *Software* Engineering**, Vol 24, No 12, Special Issue on Scenario Management, December, 1998.

ROSSON, M.B., and CARROLL, J. **Narrowing the Specification Implementation Gap in Scenario-based Design: envisioning work and technology in system development**. John Wiley and Sons, New York, 1995. p. 247-278.

RYAN, M. **Defaults in Specifications**. Proceedings of IEEE International Symposium on Requirements Engineering. IEEE Computer Society Press. San Diego, EUA. Jan. 1993.

SANTANDER, V.F.A. **Integrando Modelagem Organizacional com Modelagem Funcional**. Tese de doutorado, Centro de Informática - UFPE, dezembro 2002.

SCHNEIDER, G. and WINTERS, J. **Applying Use-*CASEs*: a practical guide**. Addison-Wesley. 1998.

SHARP, H. FINKELSTEIN, A. and GALAL, G. **Stakeholder Identification in the Requirements Engineering Process.** Workshop on Requirements Engineering Process. Florença, Itália, 1999.

SHAW, M. and GAINES, B. **Requirements Acquisition.** *Software* Engineering Journal, vol 11, 1995.

SILVA, M.A.G. **Uma Ferramenta para Apoiar a Definição deRequisitos no Desenvolvimento de *Software* Disttribuído.** Relatório Final de Projeto de Iniciação Científica. Departamento de Infomática. Univarsidade Estadual de Maringá, 2001.

SOHLENKAMP, M. **Supporting group awareness in multi-user environment through perceptualization.** Dissertation. Fachbereich Mathematik-Informatik der Universität – Gesamthochschule – Paderborn. Fevereiro, 1998.

SOMMERVILLE, I. ***Software* Engineering** - 6a edição. Addison Wesley, 2001.

SUN, METADATA Repository (MDR) home, SUN Microsystems, Inc. Palo Alto, USA, Disponível em: http://mdr.netbeans.org, acesso em: 10 jul. 2002.

SUTCLIFFE, A. A scenários-based Requeriments Analysis. **Journal of Requirements Engineering** – vol 3 – Springer Verlag, 1998. p. 48-65.

TELELOGIC AB. ***DOORS***. Disponível em http://www.telelogic.com, acesso em: 16 jun. 2003.

TORANZO, M. and CASTRO, J. **A Comprehensive Traceability Model to Support the Design of Interactive Systems.** In: International Workshop on Interactive System Development and Object Models - WISDOM 99. Lisboa, Portugal, Proceedings ..., Jun. 1999.

VILLER, S. and SOMMERVILLE, I. **Social Analysis in the Requirements Engineering Process: from ethnography to method.** In: 4 th International Symposium on Requirements Engineering, Limerick, Ireland, Proceedings ..., Jun. 1999.

WEIDENHAUPT, K., POHL, K., JARKE, M., HAUMER, P. Scenario Usage in System Development: current practice. **NIEEE *Software,*** vol 15 N. 2 , 1998. p 34-45.

WEIRINGA, R. J. **Requirements Engineering – Frameworks for Understanding.** New York, John Wiley e Sons Ltd. 1996.

WIEGERS, K. ***Software* Requirements**. Microsoft Press. 1999.

YANAGA, F. **Use Case Drawing Tool.** Projeto de Iniciação Científica. Departamento de Infomática. Universidade Estadual de Maringá, 2003.

YU, E. **Modeling Strategic Relationship for Process Reengineering**. PhD thesis, Computer Science Department, University of Toronto. Toronto, Canada. 1995.

ZAVE, P. Classification of Research Efforts in Requirements Engineering. **ACM Computer Surveys**, vol 29, n°4. 1997.

ZORMAN, L. **Requirements Envisaging Though Utilizing Scenarios – REBUS.** Ph.D. Dissertastion. University of Sourthrn California - 1995.

ANEXO 1

O PROCESSO DE CONSTRUÇÃO DO *LEL*, CENÁRIOS E *USE CASE*

A) O PROCESSO DE CONSTRUÇÃO DO *LEL* (CYSNEIROS, 2001)

O processo de construção do *LEL* do *UdI* (NETO, 2000) engloba as seguintes atividades: identificar, classificar e descrever símbolos.

O primeiro passo no processo de construção do *LEL* consiste na identificação dos símbolos. Para tal, devemos identificar as palavras ou frases da linguagem praticada no *UdI* que pareçam ter um significado especial. Em geral, estas palavras ou frases são utilizadas com freqüência por atores do *UdI* ou outras fontes de informação. Palavras ou frases que parecem sem sentido, ou fora de contexto, têm grande probabilidade de serem símbolos do *UdI*. As técnicas de elicitação mais adequadas para esta atividade são a observação, leitura de documentos e entrevista não estruturada. As heurísticas de identificação se resumem a procurar sujeitos, verbos, objetos e estados (predicativos) presentes em frases que surgem no *UdI* e que possuem um significado especial no *UdI*. Antes de utilizar as heurísticas, deve-se separar os períodos compostos em orações simples, transportar orações na voz passiva para a voz ativa e transformar as formas substantivas de verbos para a forma verbal correspondente. Observou-se que a transformação das formas substantivas de verbos para formas verbais correspondentes é extremamente importante para que se possa utilizar as heurísticas de construção de cenários a partir do *LEL* do *UdI*, propostas por HADAD (1997).

O segundo passo deste processo consiste na classificação dos símbolos. Neste passo devemos determinar se os símbolos são sujeitos, verbos, objetos e estados (predicativos) de frases que surgem no *UdI*.

A última etapa do processo resume-se à descrição dos símbolos. Para realizarmos esta etapa temos de criar entradas do *LEL* referentes aos símbolos identificados. Para criar uma entrada do *LEL* para um símbolo, deve-se descrever

noções e impactos, seguindo as heurísticas de representação. Algumas destas heurísticas de representação são mostradas no Anexo 2. A utilização destas heurísticas deve ser observada ao se construir o *LEL*, para que as heurísticas de construção de cenários a partir do *LEL*, propostas por HADAD (1997), possam ser utilizadas. Nesta atividade, identificam-se os sinônimos dos símbolos descritos por entradas do *LEL*. Eventualmente, são identificados novos símbolos nesta atividade, bem como novas noções e impactos de outros símbolos. As entradas do *LEL* assumem as mesmas classes - sujeito, verbo, objeto e estado (predicativo) – dos símbolos que descrevem.

O processo de construção do *LEL* é continuado sendo retomado sempre que surgirem alterações no *UdI* que tragam novos símbolos para o domínio ou pela descoberta de novos símbolos durante a elicitação de um dado símbolo (princípio da circularidade).

B) O PROCESSO DE CONSTRUÇÃO DE CENÁRIOS A PARTIR DO *LEL* DO *UdI* (CYSNEIROS, 2001)

HADAD propõe uma estratégia baseada no uso do *LEL* para gerar cenários (HADAD, 1997) (HADAD, 1999). Esta estratégia baseia-se em algumas heurísticas para realizar tal construção, descritas a seguir:

Passo 1: Identificar atores do *UdI*.

Identificar os símbolos do *LEL* do *UdI* classificados como sujeito. Estes símbolos representam os atores do *UdI*, os quais são classificados em primários (atores que realizam ações diretas sobre o sistema) e secundários (atores que recebem/fornecem informações para o sistema, mas não executam ações diretas sobre a aplicação).

Passo 2: Identificar cenários candidatos.

Recuperar os impactos das entradas do *LEL* do *UdI* associadas aos atores do *UdI*, os quais foram obtidos no passo 1. Eliminar os impactos repetidos. Os impactos resultantes são os títulos dos cenários candidatos. Primeiramente obtemos os impactos

das entradas do *LEL* do *UdI* associadas a atores principais e, em seguida, o mesmo é realizado para as entradas do *LEL* do *UdI* associadas aos atores secundários.

Passo 3: Descrever cenários candidatos.

Para cada cenário candidato identificado no passo anterior, faça:

- Se o título do cenário candidato possui um símbolo do *LEL* do *UdI* classificado como verbo, então:
  1. Recuperar a entrada do *LEL* do *UdI* referente ao símbolo classificado como verbo.
  2. Definir o objetivo do cenário com base no seu título e nas noções da entrada recuperada.
  3. Verificar se existe alguma ordem de precedência entre o impacto que originou o cenário com os outros impactos. Caso exista, incluindo contexto o impacto que o precede como uma pré-condição.
  4. A partir de cada impacto da entrada, definir os episódios do cenário. Os episódios podem ser reescritos, utilizando a sintaxe para episódios, que permite expressar comportamento. O mesmo não é possível nos impactos de entradas de *LELs* do *UdI*.
  5. Identificar atores e recursos, os quais devem ser entradas do *LEL* do *UdI* classificadas respectivamente como sujeito e objeto.
- Se o título do cenário candidato não possui um símbolo do *LEL* do *UdI* classificado como verbo, então:
  1. Identificar os símbolos do *LEL* do *UdI* presentes no título do cenário candidato.
  2. Definir o objetivo do cenário com base no seu título.
  3. Verificar se existe alguma relação de precedência entre o impacto que originou o cenário com outros impactos. Caso exista, inclui-se no contexto o impacto que o precede como pré-condição.
  4. Definir os atores e recursos com base nos símbolos recuperados. Os atores são entradas do *LEL* do *UdI* classificadas como sujeito. Os

recursos são entradas do *LEL* do *UdI* classificadas como objeto. Neste caso, não se definem episódios a partir do *LEL* do *UdI*.

Passo 4: Completar os cenários com informações provenientes do *UdI*.

Consiste em buscar informações adicionais, com os atores ou outras fontes de informação do *UdI*, para completar partes de cenários que não puderam ser preenchidas com as informações fornecidas pelo *LEL* do *UdI*. Em geral, este passo se aplica mais diretamente aos cenários para os quais não se definiram episódios.

Passo 5: Analisar cenários.

Consiste em validar e verificar cenários. Os cenários são validados com os clientes a fim de identificar erros, omissões e/ou ampliar informações de episódios. Esta ampliação de informações de episódios engloba a substituição de um episódio por vários episódios no mesmo cenário e a descrição de episódios através de subcenários. A tarefa de verificação de cenários consiste em:

- **Unificar cenários.** Dois ou mais cenários, que possuem os mesmos episódios ou o mesmo objetivo e contexto, são agrupados em um único cenário. Quando necessário, utiliza-se a forma condicional para descrever episódios diferentes.
- **Detectar sub-cenários.** Se, em cenários provenientes de atores principais, um conjunto de episódios corresponde a um cenário proveniente de um ator secundário, substituir aqueles episódios por este cenário.

C) O PROCESSO DE CONSTRUÇÃO DE *USE CASE* (SANTANDER, 2002)

Segundo SANTANDER (2002), um possível processo de construção de *use case* inicia com a descoberta dos **atores** do sistema e prossegue com a descoberta dos ***use cases*** relacionados com estes atores. Para cada ator, são encontrados todos os *use cases* relacionados ao mesmo. Isso ocorre porque cada ator requer do sistema algumas

funcionalidades, sendo que os passos necessários para obter estas funcionalidades são descritos através do *use case*.

O segundo passo consiste em definir os caminhos básicos (cenários primários) e posteriormente os caminhos alternativos (cenários secundários) para cada um dos *use case*. O terceiro passo envolve revisar descrições de aspectos comportamentais de *use case* encontrando relacionamentos do tipo ***<<include>>***, ***<<extend>>*** e ***<<generalization>>***. Este processo é, geralmente, realizado adotando-se o princípio de desenvolvimento de *software* iterativo e incremental (JACOBSON et al., 1999). Após definidos todos os *use case* e atores do sistema, desenvolve-se um modelo de *use case* utilizando as notações da *MDSODI* descrito no capítulo 3 Quadros 1, 2 e 4.

No entanto, a tarefa de descobrir e descrever *use case* não é tão simples, pois na maioria das situações, exige um certo grau de experiência de engenheiros de requisitos. Uma série de heurísticas para auxiliar o engenheiro de requisitos no desenvolvimento de *use case*, podem ser encontradas em SCHNEIDER e WINTERS (1998) e BOOCH et al. (1999).

ANEXO 2

HEURÍSTICAS DE REPRESENTAÇÃO DE ENTRADAS DO *LEL*

(CYSNEIROS, 2001)

- Cada entrada tem zero ou mais sinônimos.
- Cada entrada tem uma ou mais noções.
- Cada entrada tem um ou mais impactos.
- Escreva noções e impactos usando frases simples, que expressam uma única idéia.
- A descrição de noções e impactos deve respeitar os princípios de circularidade e vocabulário mínimo.
- Para uma entrada que descreve um símbolo classificado como sujeito:
    - As noções devem dizer quem é o sujeito (pessoa ou coisa) do qual se diz alguma coisa;
    - Os impactos devem registar as atividades que o sujeito realiza;
    - Os sinônimos devem conter termos que tenham o mesmo sentido do símbolo no *UdI*.
- Para uma entrada que descreve um símbolo classificado como verbo:
    - As noções devem descrever sucintamente a ação e dizer quem é o sujeito (pessoa ou coisa) que realiza a ação,quando a ação acontece e quais as informações necessárias para realizar a ação;
    - Os impactos devem registrar os procedimentos que fazem parte da ação, as condições/situações decorrentes da ação e outras ações que deverão ocorrer;
    - Os sinônimos devem registrar a forma substantivada, a voz passiva e a forma infinitiva do verbo.
- Para uma entrada que descreve um símbolo classificado como objeto:
    - As noções devem definir o ser (pessoa ou coisa) que sofre ações;

- Os impactos devem descrever as ações que podem ser aplicadas ao objeto;
- Os sinônimos devem conter termos que tenham o mesmo sentido do símbolo no *UdI*.

ANEXO 3

MDR (METADATA REPOSITORY)

A) *MDR - METADATA REPOSITORY* (MORO, 2002)

MDR ou Metadata Repository é uma implementação de um repositório de metadados baseado no padrão *MOF* para trabalhar de forma integrada à ferramenta de código aberto *Netbeans*.

A implementação do repositório de metadados é fornecido aos usuários da ferramenta *Netbeans*.

O *MDR* contém a implementação de um repositório de metadados baseado no padrão *MOF*. A interface do repositório é baseada na especificação *JMI* juntamente com algumas capacidades adicionais que ajudam a sua incorporação no ambiente de desenvolvimento da ferramenta.

Entre as propriedades do *MDR* pode-se destacar o suporte à importação e exportação de documentos no padrão *XMI* e, a geração automática de interfaces *Java* para acessar metadados descritos por metamodelos no padrão *MOF*.

Além de ter sido criado para trabalhar de forma integrada à ferramenta *Netbeans*, o *MDR* também pode ser executada individualmente e, fornece suporte à criação, ao armazenamento e à recuperação de metadados.

Como parte de sua arquitetura o MDR fornece um conjunto de interfaces que provêm métodos para indexação e persistência dos dados do repositório. Apesar de contar apenas com implementações destas interfaces para persistência em memória e, em um banco de dados próprio utilizando o sistema de arquivos e árvore B, é possível adicionar outras implementações destas interfaces para incluir um esquema de persistência alternativo baseada, por exemplo, em servidores de banco de dados relacionais através de *JDBC*.

B) *MOF - META OBJECT FACILITY* (MORO, 2002)

A especificação de *MOF* (*Meta Object Facility*) define um padrão para a definição de metadados. O *MOF* fornece o suporte à descrição de qualquer tipo de metadados que possam ser descritos utilizando-se técnicas de modelagem de objetos.

A especificação do padrão *MOF* adota a definição de que metadados é qualquer tipo de dado que descreve outro dado. Consequentemente, um modelo é definido como uma coleção de metadados que descrevem uma coleção de dados relacionados. Nesta especificação, é apresentada uma arquitetura conceitual para descrição de metadados, chamada de arquitetura *MOF*, e, um modelo, chamado de modelo *MOF*, que é na realidade uma linguagem de modelagem de objetos semelhante à *UML*.

Essa arquitetura de metadados *MOF* é apresentada como uma arquitetura de 4 camadas, onde cada camada de metadados descreve os dados da camada seguinte:

- **Camada 1**, camada de dados ou M0: é composta pelos dados, pelas informações que precisam ser descritas. A letra M de M0 diz respeito ao nível de descrição de dados, por exemplo M0 é composta por dados, já a camada M1 é composta por metadados que descrevem os dados da M0, a camada M2 é composta por meta-metadados, e assim por diante.
- **Camada 2**, camada de modelos ou M1: é composta pelos metadados que descrevem os dados da camada 1, ou seja, contém os modelos de dados enquanto que a camada 1 contém os dados.
- **Camada 3**, camada de metamodelos ou M2: é composta por metadados que descrevem os modelos de dados da camada 2, ou seja, contém metamodelos.
- **Camada 4**, camada de meta-metamodelos, ou M3: contém o modelo *MOF* que é na realidade um meta-metamodelo utilizado para descrever os metamodelos da camada 3.

Segundo a especificação *MOF*, a camada 4 contém, justamente, uma linguagem de modelagem semelhante à *UML* para ser usada na descrição destes

metamodelos e esta pode ser enxergada como uma linguagem abstrata para a definição de metamodelos.

A especificação *MOF* também define o chamado mapeamento *MOF IDL*. Esse mapeamento permite que a partir de uma especificação de um metamodelo *MOF* (um metamodelo descrito pelo modelo *MOF*), seja produzido um serviço de metadados *CORBA*. O mapeamento *MOF IDL* é um conjunto de templates padrão que fazem o mapeamento de um metamodelo *MOF* para um conjunto de interfaces *IDL CORBA* correspondente, de forma que, dado um modelo de metadados *MOF* como entrada, tem-se como resultado um conjunto de interfaces *IDL CORBA* que podem ser usadas na representação dos metadados. Essas interfaces permitem a criação, alteração e, o acesso à metadados na forma de objetos *CORBA*.

Além do mapeamento *MOF IDL*, a especificação *MOF* também fornece um conjunto de interfaces *IDL CORBA* para os objetos *CORBA* que representam um metamodelo *MOF*. Isso é possível, uma vez que o modelo *MOF* é definido utilizando-se o próprio modelo *MOF* na sua definição. Conceitualmente, o modelo *MOF* localiza-se na camada 4 ou M3 e, este estaria em conformidade com um modelo de uma camada 5 ou M4 que seria isomórfica ao modelo *MOF*. Assim, é possível aplicar, também, o mapeamento *MOF IDL* ao próprio modelo *MOF* obtendo-se o conjunto de interfaces *IDL CORBA* que é fornecido na sua especificação.

Finalmente, a especificação *XMI* é definida com base no padrão *MOF*. Na realidade, *XMI* é definida como uma tecnologia de intercâmbio de metadados baseados no padrão *MOF*. *XMI* é apresentada a seguir.

C) *XMI - XML METADATA INTERCHANGE* (MORO, 2002)

XMI consiste, basicamente, de um formato de intercâmbio de metadados. O principal propósito da tecnologia *XMI* é possibilitar a troca de metadados entre ferramentas de modelagem que utilizem *UML* e repositórios de metadados em um ambiente distribuído e heterogêneo. *XMI* melhora o gerenciamento e a interoperabilidade de metadados em ambientes distribuídos em geral e em ambientes distribuídos de desenvolvimento em particular.

De uma forma geral, *XMI* tem como base três tecnologias tidas como padrão:

- *XML - eXtensible Markup* Language criada pelo consórcio *W3C*;
- *UML - Unified Modeling Language*, um padrão de modelagem da *OMG*;
- *MOF - Meta Object Facility*, um padrão de repositório de metadados também da *OMG*.

Enquanto a *UML* define uma linguagem de modelagem orientada à objetos adotada como padrão por uma grande variedade de ferramentas de modelagem do mercado, o padrão *MOF* define um framework extensível para definição de modelos de metadados e fornece ferramentas com uma interface programável para o armazenamento e acesso de metadados em repositórios. A tecnologia *XMI* por sua vez, permite o intercâmbio de metadados na forma de cadeias de caracteres (*streams*) ou documentos em um formato padrão definido em *XML*.

Na realidade *XMI* pode ser vista como um par de mapeamentos, sendo o primeiro entre metamodelos *MOF* e *DTDs XML*, e o segundo entre metadados *MOF* e documentos *XML*.

Se considerada em um contexto mais amplo, ou seja, além do padrão *MOF*, *XMI* pode ser vista como um formato de intercâmbio de metadados independente da tecnologia de middleware. Ela pode ser utilizada na troca de metadados entre qualquer ferramenta e repositório que seja capaz de codificar e decodificar documentos *XMI*. Isso significa que, apesar de existir um forte relacionamento entre *UML*, *MOF* e *XMI*, a este último fornece uma tecnologia a ser usada no intercâmbio de metadados que não precisam ser baseados no padrão *MOF*, é necessário apenas que exista um mapeamento entre o metamodelo dos metadados e o documento *XMI*.

A especificação do padrão *XMI* contém um conjunto de regras de produção para a geração de definições *DTD* à partir de metamodelos *MOF* e, um conjunto de regras de produção para geração de documentos *XML* baseados em metadados *MOF*.

D) *JMI - JAVA METADATA INTERFACE* (MORO, 2002)

A especificação de *Java* Metadata Interface (*JMI*) define uma infra-estrutura dinâmica e neutra com respeito à plataforma, que possibilita a criação, armazenamento, descoberta e intercâmbio de metadados. *JMI* baseia-se na especificação *MOF* da *OMG* e define um mapeamento *Java* para o padrão *MOF*.

Enquanto *MOF* fornece um conjunto de construções para definição de metadados juntamente com um mapeamento destas construções para *IDL CORBA*, *JMI* fornece um mapeamento *Java* para *MOF* definindo dessa forma um modelo de programação *Java* para acesso à metadados.

Utilizando *JMI*, por exemplo, é possível que ferramentas com modelos baseados em *UML*, em acordo com o padrão *MOF*, tenham as interfaces *Java* para seus modelos geradas automaticamente. Além disso, o intercâmbio de metamodelos e metadados é possível através do suporte que *JMI* tem para o padrão *XMI* .

Através da tecnologia *JMI*, qualquer aplicação *Java* pode obter acesso à metadados e metamodelos em conformidade com o padrão *MOF*. Além disso, da mesma forma que *MOF* especifica um conjunto de regras para geração automática de *IDLs CORBA*, *JMI* também específica um conjunto de regras que permitem a geração de um conjunto de interfaces de programação *Java* para a manipulação das informações específicas de uma instância de um metamodelo.

De uma forma geral, *JMI* fornece ao desenvolvedor *Java* pelo menos os seguinte benefícios:

- Um framework para metadados que fornece um modelo de programação *Java* para acesso aos metadados;
- Um framework para integração e interoperabilidade de aplicações *Java*;
- Integração com a arquitetura de metadados e modelagem da OMG.

Espera-se que implementações da especificação *JMI* forneçam uma infra-estrutura de gerenciamento de metadados que facilite a integração de aplicações, ferramentas e serviços.